Proyecto de Sistemas Informáticos.
Curso 2008-2009.

Paralelización con CUDA de algoritmos de verificación facial.

Componentes del grupo:
Alfonso García Pérez
Guillermo Hernández González
Daniel Tabas Madrid

Directores del proyecto:
Jose Ignacio Gómez Pérez
Christian Tenllado van der Reijden

Facultad de Informática.
Universidad Complutense de Madrid.

Paralelización con CUDA de algoritmos de verificación facial.

Memoria de proyecto fin de carrera presentada por Alfonso García Pérez, Guillermo Hernández González y Daniel Tabas Madrid en la Universidad Complutense de Madrid, realizado bajo la dirección de Jose Ignacio Gómez Pérez y Christian Tenllado van der Reijden.

Madrid, a 2 de julio de 2009.

Autorización

Autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales, y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y o el prototipo desarrollado.

Alfonso García Pérez

Guillermo Hernández González

Daniel Tabas Madrid

A nuestras familias, novias, amigos, en definitiva,
a toda la gente que nos ha apoyado para que este
proyecto saliera adelante.

Agradecimientos

Queremos dar las gracias a Christian y a Nacho, por no filtrar nuestros cientos de e-mails como spam. También queremos agradecer a todo el equipo de técnicos de Artecs, en especial a Adrián, Ezequiel, Luis y Roberto, por su ayuda incluso fuera de horas de trabajo. Agradecer también a Héctor por su ayuda con FaVeS, y por comentar tan bien su código.

Resumen

Los delitos y fraudes derivados de la suplantación de identidad generan pérdidas millonarias para empresas y naciones. Hoy en día existen diversos métodos biométricos para la verificación de identidad. Entre ellos se encuentra la verificación facial, de gran interés práctico por su carácter no intrusivo.

Los algoritmos que se aplican para la verificación facial tienen un alto coste computacional, dificultando su uso en aplicaciones de tiempo real. Sin embargo estos algoritmos presentan un alto grado de paralelismo a nivel de datos que podría explotarse con plataformas multicore.

En la actualidad uno de los principales exponentes de las plataformas multicore son las unidades de procesamiento gráfico (GPU). En este proyecto se ha abordado la implementación de diversos algoritmos de verificación facial en GPU. Los resultados en términos de rendimiento han sido altamente satisfactorios, llegando a obtenerse speedups superiores a 200 en comparación con implementaciones paralelas tradicionales (OpenMP). Asimismo se ha desarrollado una interfaz gráfica que permite realizar la verificación de la identidad de una persona a partir de dos fotografías con cualquiera de los métodos implementados.

Palabras clave: CUDA, NMF, reconocimiento facial, paralelización, biometría, GPU, KDA, Gabor, SIMT, stream processing.

Abstract

Crime and fraud derived from identity theft produce loss of millions to enterprises and nations. Nowadays there exist several biometric methods for identity verification. One of them is facial recognition, of great practical interest due to its non-intrusive character.

The algorithms applied to facial verification demand high computational cost, making it difficult to use them in real-time applications. However, these algorithms show a large degree of data-level parallelism which could be exploited with multi-core platforms.

One of the main current representatives of multi-core platforms are graphics processing units (GPUs). This project deals with the implementation of several face verification algorithms in GPUs. The performance results were highly satisfactory, reaching speedups of 200 when compared to traditional parallel implementations (OpenMP). Furthermore, a graphical interface that allows performing identity verification of a person with any of the implemented methods was developed.

Key words: CUDA, NMF, facial recognition, parallelization, biometrics, GPU, KDA, Gabor, SIMT, stream processing.

Índice general

Capítulo 1

Introducción

En la sociedad actual la verificación de la identidad es un problema extendido. Delitos basados en la usurpación de la identidad se multiplican en las dos últimas décadas: inmigración ilegal, tratar de mantener la confidencialidad empresarial, etc. Tanto las empresas como las naciones mantienen diversos métodos de seguridad para mantener a raya estas prácticas. Existe, por ejemplo, la problemática del control de fronteras. Diversas malas prácticas, tanto relacionadas con la usurpación de la identidad como no, deben ser vigiladas especialmente. Por ello existe el pasaporte, como un sistema válido para identificar al individuo entre naciones. El pasaporte contiene información de los datos personales, autenticidad del mismo pasaporte y, además, una fotografía. Normalmente, en fronteras o aduanas la parte más usada para la identificación es la imagen, debido a que es un método rápido y sencillo de verificar la identidad de una persona. Podemos ilustrar la idea pensando, por ejemplo, que difícilmente se podría parar a todos los viajeros en un aeropuerto para mantener una serie de preguntas a fin de asegurar la identidad.

Desgraciadamente, la mayoría de las medidas resultan ineficaces en muchos sentidos. Pueden ser documentos falsificables, o que si son robados y perdidos pueden ser usados por otras personas sin ningún tipo de verificación, como por ejemplo, una tarjeta de crédito en compras por internet.

1.1. Métodos biométricos

Motivados por todo lo anterior se han desarrollado métodos de identificación fiables basados en la complexión no cambiante de los seres humanos como por ejemplo huellas, ojos, distancia o forma de las partes de la cara. Se conocen como métodos biométricos. Podemos enumerar los métodos usados: las huellas dactilares, el reconocimiento del iris y el reconocimiento facial. El iris es un músculo dentro del ojo que regula el tamaño de la pupila, controlando la cantidad de luz que entra en el ojo. Es la porción coloreada del ojo, que basa su color en la cantidad del pigmento melatonina dentro del músculo. Aunque la coloración y la estructura del iris están genéticamente ligadas, los detalles de los patrones no lo están. Esto hace que el iris de cada persona sea distinto y lo convierte en un método fiable de identificación. Se trata de obtener una imagen clara del ojo de la persona y ampliarla, buscando el patrón que define el iris respecto al tamaño de la pupila y la imagen.

La identificación por huella dactilar es uno de los métodos biométricos más conocidos y publicitados. Lleva siendo usada más de un siglo y ha sido mejorada por organismos que la han considerado de gran importancia como el FBI o la CIA. Para documentación

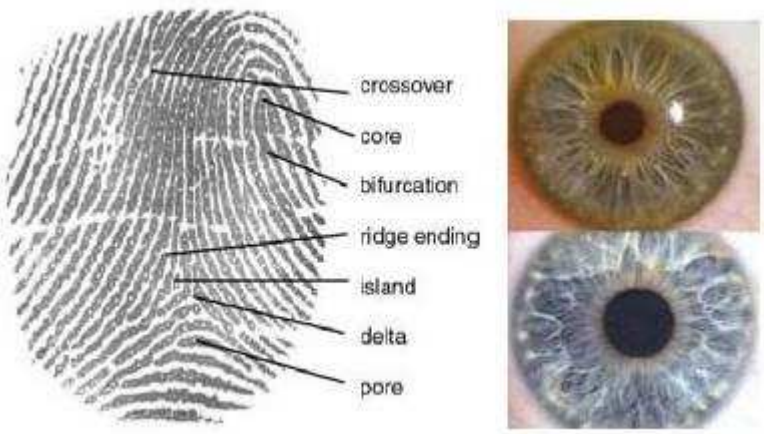


FIGURA 1.1: Comparación de dos iris

oficial es fácil de conseguir y es más segura puesto hay diez dedos de los cuales se puede tomar la información. La identificación por huella dactilar está basada principalmente en las minucias, o la ubicación y dirección de los finales y bifurcaciones de los surcos a lo largo de su trayectoria por el dedo.

Estos métodos, aunque fiables, resultan intrusivos ya que requieren la asistencia de la persona para poder llevar a cabo la identificación. No es fácil tratar de obtener la imagen clara del iris de una persona sin su colaboración, y es una práctica digna de detectives el conseguir huellas dactilares de la persona y más aún saber de qué dedo son. Por ello sería interesante potenciar al tercero de los métodos ya que es el más fácilmente implementable.

1.2. Verificación Facial

La utilización del rostro para distinguir a los seres humanos es la práctica más sencilla y natural; sin embargo es también la menos fiable. La cara de los seres humanos esta llena de formas y distancias que, en unión, hacen el rostro de un ser humano único, pero que hacen que la verificación facial sea el más complicado de los mecanismos de identificación. Pequeños rasgos, como por ejemplo la distancia de los ojos, se pueden repetir entre personas, pero entonces no tendrán el mismo arco de la nariz. Todo esto requiere un análisis más profundo y complejo de las formas de la cara para poder realizar una verificación fiable. La fiabilidad del resultado es dependiente del método usado.

La verificación facial se puede clasificar de varias formas. Una posible clasificación es:

- Basada en la geometría: estudia de forma puntual las distancias y formas de lugares puntuales del rostro para obtener la distinción geométrica del individuo respecto de los demás.
- Basada en texturas: analiza las sombras, colores y profundidad de la imagen.

Otra clasificación es:

- Métodos holísticos: realizan la identificación a través del análisis de la imagen en su conjunto.

- Métodos locales: sólo usan una serie de puntos de la imagen para realizar la identificación.

¿Por qué usar este método si es menos fiable y más complicado? Sencillamente porque es más sencillo conseguir los datos necesarios. La cámara fotográfica es un objeto común en la actualidad y la mayoría de la gente tiene una o incluso puede usar su teléfono móvil como tal. Esto hace que conseguir imágenes sea muy sencillo con la colaboración o no del individuo y por lo tanto es un método óptimo para muchas aplicaciones de seguridad.

El reconocimiento facial, lejos de ser perfecto ahora mismo, ha mejorado mucho en los últimos tiempos: métodos automáticos de identificación mejorados, tratamiento más correcto de las imágenes para su análisis, conocimiento más perfecto sobre cómo organizar bases de datos de caras, capacidad para conseguir y almacenar colecciones de imágenes de gran resolución, etc.

Conseguir métodos computacionales que realicen el reconocimiento no es sencillo y requiere mucho estudio obtener resultados satisfactorios. Por otro lado, se presenta el problema del rendimiento. La creación de bases de datos de miles de imágenes o la comparación de una imagen con todas las que hay almacenadas en una de ellas, incluso para una CPU moderna, puede llevar demasiado tiempo como para que puedan realizarse en tiempo real.

1.3. Demanda computacional

Debido al uso que se da al reconocimiento facial, es necesario que se pueda hacer en tiempo real. Por ejemplo, en una aduana sería algo impensable y absurdo que hicieran esperar a cada pasajero 3 minutos para poder verificar su identidad y dejarle pasar. Necesitamos por lo tanto rapidez de tiempo real. ¿Podemos conseguirlo?

Estos algoritmos realizan transformaciones típicas de tratamiento de imágenes: multiplicación de matrices, búsqueda de autovalores o autovectores, divisiones o multiplicaciones punto a punto, etc. Aunque las operaciones de una o dos imágenes puedan parecer asequibles para la capacidad de procesamiento de un computador actual, no podemos decir lo mismo de operaciones de cientos o miles de imágenes. Matrices de gran tamaño necesitan una gran cantidad de poder computacional.

Debemos buscar una forma de acelerar el cálculo. Esto, en teoría, se podría realizar de dos formas: usar variaciones del método o mejorar la programación del método para que sea más eficiente. Vamos a optar por la segunda opción dado que en la actualidad se han desarrollado diversos mecanismos para tal propósito. Uno de los estos mecanismos es la paralelización, es decir, la división de los cálculos en distintas partes con las que se pueda trabajar a la vez.

Una CPU normal en la actualidad ya no suele poseer un solo procesador, ahora se fabrican con dos, cuatro o incluso ocho; aun así no es capaz de acelerar suficientemente el proceso. Por ejemplo, para realizar la multiplicación de dos matrices de 10.000 por 10.000 en tiempo real tendríamos un requisito computacional de Gflops. El rendimiento de cualquier CPU queda lejos de tales cifras, por lo cual necesitamos una forma para conseguir más capacidad de computación. Se podría formar un cluster de computadores con el que conseguir los suficientes procesadores para realizar estos cálculos en tiempo real, sin embargo, es una alternativa costosa en varios sentidos: coste financiero, coste espacial, coste de mantenimiento, coste de distribución. Esta situación hace necesaria para el empresario o usuario normal la búsqueda de alternativas a un cluster habitual.

Podemos encontrar en la mayoría de los computadores de sobremesa justo un elemento que cumple ese diseño: la unidad de procesamiento gráfico (o GPU de ahora en adelante) [GPU]. El diseño moderno de las GPUs se basa en la implementación de gran cantidad

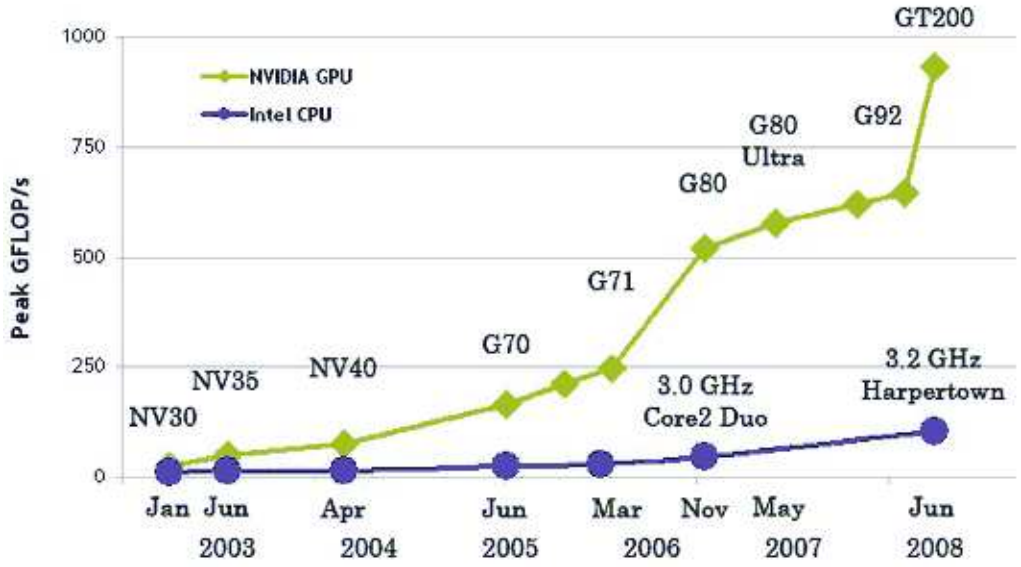


FIGURA 1.2: Comparación de capacidad de procesamiento de CPU y GPU

de unidades funcionales; estas GPUs suelen contener unos 256 procesadores de media, para el tratamiento paralelo de los gráficos de computador. Esta arquitectura coincide exactamente con lo que buscamos, ya que está diseñada para tratar gran cantidad de datos similares y divisibles de forma paralela.

En la figura 1.2 podemos observar la distinta capacidad de cálculo que han ido desarrollando en el tiempo las CPU y las GPU.

Por lo tanto, para conseguir gran cantidad de procesadores y poder paralelizar será interesante el uso de la GPU. En la actualidad, la mayoría de los fabricantes ponen a disposición del programador los manuales y compiladores, lo que permite programar para la GPU. Por lo tanto, podemos disponer del útil físico y el método de programación de forma sencilla y mucho más barata que otras alternativas.

1.4. Objetivos

Los objetivos de este proyecto serán principalmente:

- Implementación eficiente de métodos existentes de verificación facial. En muchos casos, nos basaremos directamente en el método matemático, pero tampoco se descarta la transformación de programas ya existentes al paradigma de programación basado en flujos.
- Medir el rendimiento de los diversos métodos tras su implementación y optimización. Observar cómo cambia el rendimiento de los métodos para distintos tamaños de datos iniciales y matrices y buscar cuál es óptimo en cada momento para las necesidades que se puedan presentar, como por ejemplo respecto a la variación de la base de datos inicial.
- Cuidar la robustez del método, es decir, vigilar cómo es de fiable; tratar de ver cuál genera más aciertos y se aproxima más a las exigencias.

- Comparativa de los métodos implementados con otras implementaciones paralelas no relacionadas. Poder observar de esta forma la eficiencia de ese tipo de paralelización respecto a otros modelos existentes como Open Multi-Processing (OpenMP) [Ope].
- Crear una aplicación que, usando estos métodos, nos permita ilustrarlos de forma visual. Será una interfaz de uso sencillo y visual, para que el usuario pueda fácilmente comparar dos fotos usando el método que prefiera.

Capítulo 2

Verificación Facial

Existen gran variedad de métodos de verificación facial a partir de imágenes 2D, que podemos dividir en diferentes categorías como vimos anteriormente.

En nuestro caso, nos hemos centrado en los métodos basados en texturas y holísticos como son el Gabor-*Kernel Discriminant Analysis* (también referido como Gabor-KDA para abreviar) y *Non-negative matrix factorization* (también llamado NMF). Otros dos métodos de este mismo tipo y que resultan interesantes, dado que son sobre los cuales se ha desarrollado Gabor-KDA, son *Principal component analysis* (o PCA) y *Linear Discriminant Analysis* (abreviado como LDA).

Todos estos métodos se componen de dos fases:

- Entrenamiento u *Offline*: En esta fase se busca una base sobre la cual poder realizar operaciones. Tomando las distintas imágenes y operando sobre ellas, como veremos en las siguientes secciones, podemos conseguir una base sobre la cual poder proyectar nuestras nuevas imágenes.

- Comparación u *Online*: En esta fase se proyecta la imágenes sobre la base que hemos conseguido en la fase *offline*. Se aplican métricas sobre la imagen proyectada: coseno de los ángulos, distancia euclídea, etc. Comparando los resultados y teniendo en cuenta un umbral fijado, el algoritmo da una respuesta de si la imagen es del mismo individuo o no.

2.1. Principal Component Analysis, PCA

El análisis de componentes principales, o PCA [Jol02] [Shl05], es una técnica útil para la reducción de dimensionalidad y para encontrar patrones en datos de grandes dimensiones. Matemáticamente, PCA construye una transformación lineal que escoge un nuevo sistema de coordenadas para el conjunto de datos original, en el cual la dirección que varía más es capturada en el primer eje (llamado también el Primer componente principal, denominaciones de las cuales PCA obtiene su nombre), la segunda de más variación en el segundo eje, etc. Si tenemos un grupo de k muestras, cada una de l variables aleatorias F_j PCA permite encontrar un número de factores $r < l$ que explican de forma aproximada el valor de las l variables de cada muestra. De esta forma, encontramos los componentes principales de cada muestra, que son esos r valores, en los cuales podemos proyectar las muestras; además, hemos obtenido una compresión desechando los datos de menor varianza.

PCA asume que el sistema que analizamos es lineal, que la media y la covarianza resultan importantes para el sistema y que las varianzas mayores son las más representativas del sistema. Hay dos tipos de aplicaciones de PCA, método de la covarianza y método de correlación. El primero se usa sobre datos homogéneos y el segundo sobre datos no homogéneos.

Matemáticamente, nuestro objetivo es transformar un grupo de datos X de dimensión M en un conjunto alternativo Y de dimensión L con $L < M$. Supongamos que tenemos datos comprimiendo un conjunto de observaciones de M variables, y queremos reducirlo de forma que cada observación se pueda describir con tan sólo L variables, con $L < M$. Esperamos tener los datos organizados en un conjunto de N vectores de datos $X_1 \cdots X_N$ con cada X_n representando una observación de las M variables. Calculamos la media empírica para cada dimensión $m = 1, \cdots, M$. Usamos los valores calculados de la media en un vector u de medias de dimensiones $Mx1$.

$$u[m] = \frac{1}{N} \sum_{n=1}^{N} X[m, n] \tag{2.1}$$

Calculamos la desviación de la media; para ello restamos cada vector u de la media empírica de cada columna de la matriz de datos X y guardamos los datos en una matriz B de dimensión MxN.

$$B = X - uh \tag{2.2}$$

donde h es un vector $1xN$ de todo unos.

Buscamos la matriz de covarianza C de MxM formada por la multiplicación exclusiva de B por sí misma:

$$C = E[B \otimes B] = E[B \cdot B^*] = \frac{1}{N} \sum B \cdot B^* \tag{2.3}$$

donde E es la media. Calculamos la matriz V de autovectores que diagonaliza la matriz de covarianza C:

$$V^{-1} C V = D \tag{2.4}$$

donde D es la matriz diagonal de autovalores de C.

Reordenamos las columnas de autovectores de la matriz V y los autovalores de la matriz D en orden decreciente. Los autovalores representan la distribución de la energía de los datos fuente a través de cada uno de los autovectores, donde cada autovector forma parte de una base de los datos. Usamos las primeras L columnas de V en la matriz W de MxL:

$$W[p, q] = V[p, q] \tag{2.5}$$

para $p = 1, \ldots, M$ donde $1 \leq L \leq M$ como nueva base de la matriz. Usamos el vector g como una guía para elegir el valor apropiado de L. La meta es determinar el menor valor posible de L para alcanzar un valor razonablemente alto de g, es decir, toma un porcentaje muy alto del valor total de la energía.

Para poder trabajar con imágenes, primero debemos representarlas de forma que podamos trabajar sobre ellas. Podemos tomar cada imagen como un rectángulo de tamaño NxM y representarla como un vector de dimensión NxM.

$$X = (x_1 x_2 x_3 \ldots X_{NxM}) \tag{2.6}$$

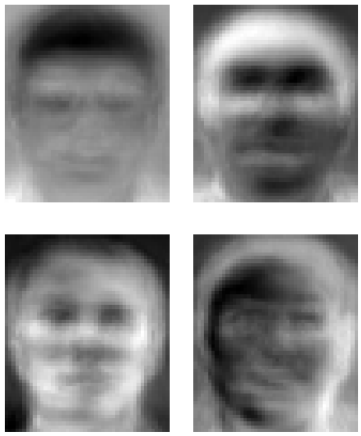


FIGURA 2.1: Ejemplo de descomposición de un rostro en eigenfaces

Es decir, hemos colocado cada píxel de la imagen uno tras otro para formar una línea o vector. Supongamos que teníamos 20 imágenes. Cada una de N píxeles de alto y M de ancho. Para cada imagen podemos crear un vector cómo hemos descrito arriba. Podemos unirlas todas para formar una gran matriz de imágenes, de forma que cada imagen es una fila de la matriz. Esto nos da el punto de partida para un análisis usando PCA. Después de usar PCA, tenemos nuestros datos originales representados por autovectores de la matriz de covarianza. ¿Cómo podemos usarlo? En el caso de reconocimiento facial, el caso de este proyecto, tendríamos imágenes originales de caras de gente. Entonces, el problema es, dada una nueva imagen, ¿esta imagen es similar a otra que tenemos? Lo que hacemos es comparar la diferencia entre imágenes, no según las originales, sino tomando como ejes de comparación los obtenidos en el análisis PCA. Esto funciona mucho mejor que comparar las imágenes según sus ejes originales, dado que PCA nos hace una análisis en términos de diferencias y similitudes al encontrar los patrones que definen caras.

Para ilustrar el resultado de todo lo anterior, podemos observar en la figura 2.1 las distintas descomposiciones, llamadas *eigenfaces*, según diferencias y similitudes que se podrían obtener al aplicar PCA sobre un rostro.

2.2. Linear Discriminant Analysis, LDA

LDA [Fis36] [Fri89] es una técnica de aprendizaje supervisado para clasificación de datos. El objetivo con LDA es obtener una proyección de los datos en un espacio de menor dimensionalidad que los datos entrantes, con el fin de que la separabilidad de las clases sea la mayor posible. Es supervisada, ya que para poder encontrar una proyección debemos entrenar el sistema con patrones etiquetados.

Matemáticamente, existen distintas implementaciones de LDA; una de ellas es Fisher-LDA, con la cual realizamos el siguiente cálculo teórico. Queremos encontrar la matriz w de proyección, que proyecte los datos de un espacio de manera que tengamos la mayor separabilidad entre sus clases.

Tenemos $x_1 \dots x_n$ patrones multidimensionales etiquetados en c clases. Cada clase tiene N_c patrones. Se busca W matriz de autovalores, para obtener $y_i = w^T x_i$ proyecciones de los patrones. Con Fisher-LDA se busca maximizar la función objetivo:

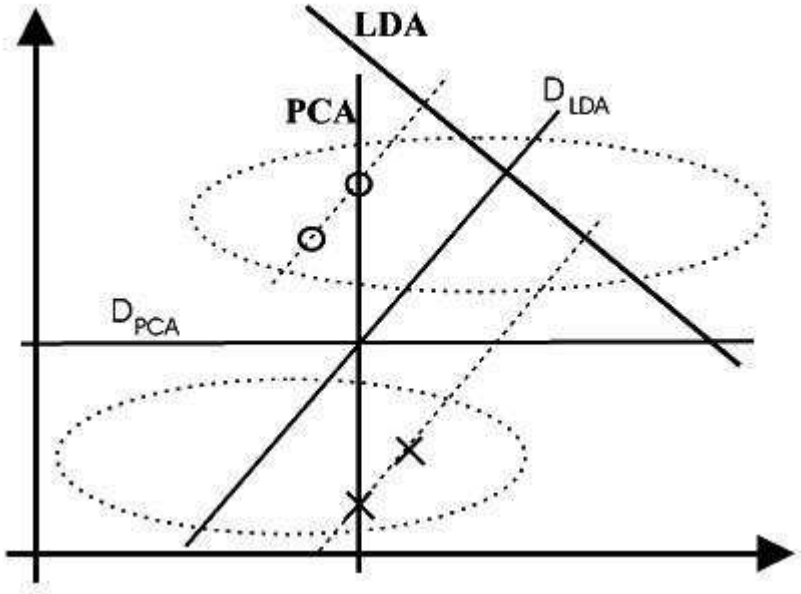


FIGURA 2.2: Clasificaciones formadas por PCA y LDA dadas una serie de distribuciones

$$J(w) = \frac{w^T S_B W}{w^T S_W W} \tag{2.7}$$

donde S_B es la matriz de dispersión interclase y S_w es la matriz de dispersión intraclase. Precisando más:

$$S_B = \sum_c N_c(\mu_c - \mu)(\mu_c - \mu)^T \tag{2.8}$$

$$S_W = \sum_c \sum_{i \in c} (x_i - \mu_c)(x_i - \mu_c)^T \tag{2.9}$$

Siendo μ_c la media de cada clase, μ la media de todos los datos, N_c la cantidad de patrones de la clase c.

Buscamos encontrar la matriz W de proyección que maximice el "cociente" entre la matriz de dispersión interclase y la matriz de dispersión intraclase. Operando se puede observar que la W que maximiza la función objetivo debe cumplir:

$$S_B W = \lambda S_W W \tag{2.10}$$

Supongamos que hacemos una descomposición de las imágenes análoga a la que se realizó en PCA para aplicar el método LDA. Al igual que en PCA necesitamos usar una base de imágenes para crear el entrenamiento básico, y, como dijimos al principio, esta vez también tendremos que darle nosotros los patrones que usará para, por así decirlo, distinguir rasgos faciales. Nuestro objetivo con LDA es que la técnica encuentre los distintos rasgos que clasifican los individuos y los lleve a distintas clases. Para comparar si dos imágenes corresponden a la misma persona, buscará la coincidencia de sus rasgos en las clases.

En la figura 2.2 se observan las distintas bases formadas por PCA y LDA respecto a las mismas distribuciones (círculos y equis). Se observa que PCA no maximiza la distancia entre las clases mientras que LDA sí lo hace. Para esta distribución en particular, PCA es mejor, dado que, en este caso en concreto, LDA incluiría ciertas proyecciones de la clase de los círculos en las equis.

2.3. Gabor-KDA

Gabor-KDA [Liu06] (*Kernel Discriminant Analysis*), desarrollado por Chengjun Liu, es un nueva técnica de reconocimiento de patrones. Se basa en la representación de imágenes de Gabor y el método de KDA. La representación de imágenes de Gabor, obtenida aplicando el método de Gabor a una representación de imagen, captura las propiedades visuales tales como localización espacial, orientación o frecuencia espacial. El método KDA es un método parecido a PCA y LDA que permite encontrar distinciones de clases en espacios no lineales.

La representación de Gabor se consigue a partir de los wavelets de Gabor. Estas wavelets reciben su nombre de Dennis Gabor, quien los propuso en 1946 para el tratamiento de señales unidimensionales. Están relacionados con los procesos en la corteza visual. Presentan propiedades interesantes, como el hecho de estar localizadas tanto en el dominio espacial como de frecuencias, y minimizar la relación de incertidumbre entre posición y frecuencia espacial. Por ésto se propusieron como método óptimo para descomposición de señales en información, para su posterior tratamiento. Una wave de Gabor es un conjunto de ondas gaussiano, es decir, una exponencial compleja, de frecuencia dada, con una envolvente gaussiana que la localiza espacialmente. Por ejemplo, para el caso bidimensional, la expresión de un wavelet de Gabor sería:

$$\psi_{\mu,\nu}(\vec{r}) = \frac{\|\vec{k}_{\mu,\nu}\|^2}{\sigma^2} e^{-\frac{\|\vec{k}_{\mu,\nu}\cdot\vec{r}\|^2}{2\sigma^2}} \left[e^{i\,\vec{k}_{\mu,\nu}\cdot\vec{r}} - e^{-\frac{\sigma^2}{2}} \right] \tag{2.11}$$

donde

σ_x, σ_y definen la función bidimensional gaussiana, los componentes del vector de ondas k_x, k_y definen la onda plana, $\vec{k}_{\mu,\nu} = k_\nu(cos\phi_\mu, sin\phi_\mu)$ y $\vec{r} = (x, y)$. Además se impone una relación entre el tamaño de la función gaussiana y la frecuencia de la onda planar de forma que todos los conjuntos de ondas tienen la misma energía $\sigma_x = \sigma_y = \frac{\sigma}{k}$.

El muestreo discreto de estas funciones wavelet es llamado el wavelet kernel, y si puede ver como un filtro bidimensional del mismo tamaño que las imágenes.

La representación de Gabor de una imagen es la convolución de la imagen con una familia de kernels de Gabor. Siendo I nuestra imagen, $I(x, y)$ la distribución en grises de la imagen y un kernel de Gabor $\psi_{\mu,\nu}$, tendremos que:

$$\mathcal{O}_{\mu,\nu}(x, y) = I(x, y) \odot \psi_{\mu,\nu}(x, y), \tag{2.12}$$

Donde $\odot$ es el operador de convolución y $\mathcal{O}_{\mu,\nu}(x, y)$ es la convolución que corresponde al kernel de Gabor en la orientación μ y escala ν. El conjunto $\mathcal{S} = \{\mathcal{O}_{\mu,\nu}(x, y) : \mu \in \{0, \ldots, 7\}, \nu \in \{0, \ldots, 4\}\}$ forma la representación de la imagen en wavelets de Gabor. Véase que la representación tiene valores complejos y que en su conjunto aumenta la dimensión de la imagen. Por esto en algunos casos, se suele submuestrear cada $\mathcal{O}_{\mu,\nu}(x, y)$ por un factor ρ.

KDA [yKM99],*Kernel Discriminant Analysis*, fue diseñado para extraer las características discriminantes de espacios no lineales. En estos tipos de espacios, es difícil encontrar de forma directa las características discriminantes de las clases del espacio. El método KDA

similar a LDA o PCA se basa en el uso del llamado *Kernel trick*, el cual aplica una función de asignación no lineal del espacio original a un espacio lineal de alta dimensionalidad en el cual podemos encontrar las características discriminantes de las clases proyectadas en el nuevo espacio. En el nuevo espacio imágenes de características parecidas pertenecerán a la misma clase e imágenes con características diferentes a una clase distinta.

El paso de imágenes a Gabor-KDA es similar que los métodos usados en PCA y LDA. Las diferencias observables al final de los 3 métodos tras su parte offline y online serán que obtendremos distintas Curvas ROC tras el entrenamiento, distintos umbrales y las imágenes a proyectar tendrán distintos resultados en cada uno de los métodos. Esto hace que, aunque tengan un diseño parecido, obtengamos unos resultados distintos para la comparación de imágenes en cada uno de los métodos, aunque usemos la misma base de caras para el entrenamiento.

2.4. Non-negative matrix factorization

NMF [Gui02] [LS99] es un algoritmo de factorización de matrices propuesto originalmente por Lee et al para el análisis de imágenes faciales. Esta técnica se puede aplicar al análisis exploratorio de datos como método de proyección para reducir la dimensionalidad de los datos o para descubrir patrones ocultos, aunque su aplicación más extendida es para facilitar la interpretación de los datos. Mediante NMF podemos reproducir de forma aproximada una matriz de datos,$V \in R^{NxM}$, como un producto de dos matrices

$$W \in R^{nxk} \tag{2.13}$$

$$H \in R^{kxm} \tag{2.14}$$

En este contexto, W representa un conjunto reducido de k factores (vectores base), mientras que H contiene los coeficientes de la combinación lineal de factores necesaria para la reconstrucción de V. A este conjunto de coeficientes se les denomina vectores codificante.

Adicionalmente, se suelen imponer las siguientes restricciones:

$$k << n \tag{2.15}$$

V,W y H no tienen valores negativos. Las columnas de W están normalizadas (la suma de los elementos de cada columna vale 1). Una de las aplicaciones que tiene este algoritmo en bioinformática es el análisis de expresión de genes, donde V representa una matriz con la expresión de n genes en m experimentos. En este caso, a las k columnas de W se les denomina experimentos base, mientras que cada fila de H representa un gen base. La mayor diferencia entre NMF y otras técnicas de factorización que han sido aplicadas al análisis de expresión de genes, tales como análisis de componentes principales (PCA), descomposición de valores singulares (SVD) o análisis de componentes independientes (ICA), radica en la restricción impuesta a los vectores base (W) y a los codificantes (H) de no utilizar valores negativos. Gracias a esta restricción se obtiene una representación de los datos basada en partes o secciones, ya que sólo se permiten combinaciones aditivas, nunca sustractivas. De esta manera, los factores pueden ser interpretados como partes de los datos o, dicho de otro modo, como elementos que tienden a aparecer juntos. Por el contrario, otras técnicas de factorización como las mencionadas anteriormente, permiten que los valores de W y H tengan cualquier signo, lo que conlleva a cancelaciones complejas de elementos positivos y negativos durante la reconstrucción del conjunto original de datos. En otras palabras, NMF

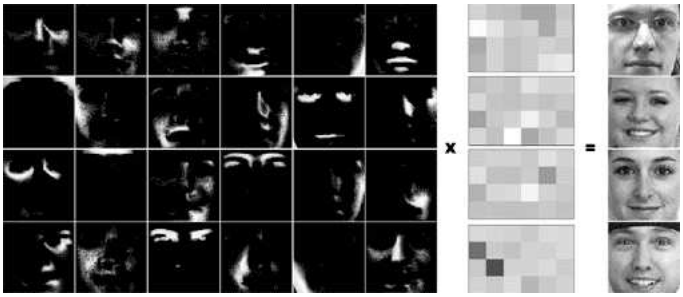


FIGURA 2.3: Descomposición de 4 rostros en NMF

tiende a producir factores que permiten una interpretación contextual relativamente sencilla, mientras que aquellos obtenidos mediante las otras técnicas no contienen un significado intuitivo. Ahora procederemos a explicar un ejemplo concreto de la aplicación del método NMF en el ámbito de la bioinformática. Para ello utilizaremos la aplicación bioNMF (disponible en http://bionmf.dacya.ucm.es/). En este caso el ejemplo es de minería de texto. La matriz de datos original contiene 24 genes de levadura de cerveza que corresponden a dos grupos funcionales diferenciados. Si introducimos los datos en el programa, y marcamos grado de factorización k=2 para distinguir ambos grupos funcionales, al cabo de 10 iteraciones obtenemos el siguiente resultado: Podemos observar los 24 genes en las filas, separados en los dos grupos (uno coloreado de azul y otro de rojo).

Este grupo de algoritmos se usa generalmente en análisis estadístico multivariante de la siguiente forma: dado un grupo n-dimensional de vectores de datos que se colocan en columnas formando una matriz n x m, se descompone en dos matrices, W de tamaño n x r y H de tamaño r x m. El factor r elegido suele ser de varios órdenes de magnitud más pequeño que n o m, por lo tanto W y H son más pequeñas que la matriz original n x m. De esta forma se puede comprimir la información contenida en la matriz original. Se puede decir que $v \approx Wh$, donde v y h son las correspondientes columnas de V y H, de lo que se puede extraer que cada vector de datos de V se aproxima mediante una combinación lineal de columnas de W multiplicadas por unos pesos, que son los componentes de h. Son algoritmos basados en actualizaciones iterativas de estas matrices, que se multiplican por un factor dependiendo de la calidad de la aproximación. Estas actualizaciones garantizan una convergencia hacia un óptimo local de la factorización de la matriz. En concreto el método de actualización que hemos utilizado en nuestra implementación es el que se basa en que la distancia euclídea ||V - WH|| no crece durante las reglas de actualización. Estas reglas son:

$$H_{a\mu} \longleftarrow H_{a\mu} \frac{\left(W^T V\right)_{a\mu}}{\left(W^T W H\right)_{a\mu}} \tag{2.16}$$

$$W_{ia} \longleftarrow]W_{ia} \frac{\left(V H^T\right)_{ia}}{\left(W H H^T\right)_{ia}} \tag{2.17}$$

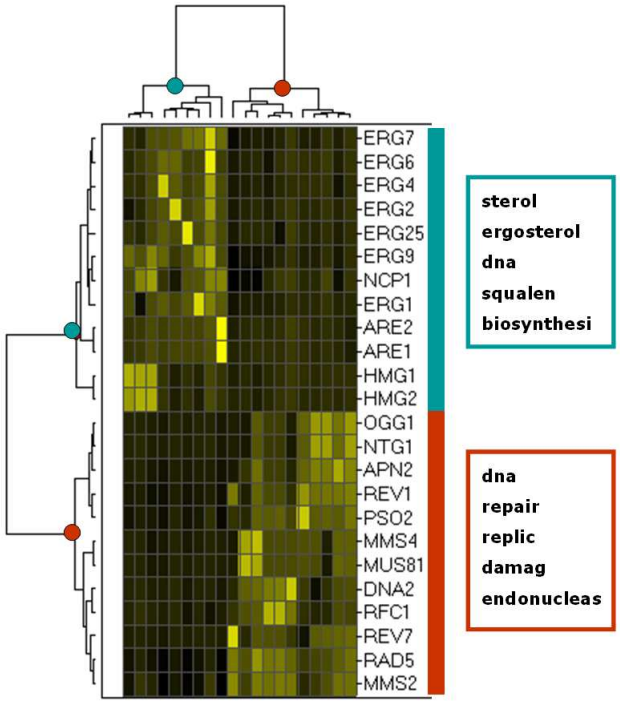


FIGURA 2.4: Nmf en la expresión de genes

Capítulo 3

GPU

La tarjetas gráficas aparecen en torno a los años 60, cuando se comienzan a utilizar los monitores como elementos de visualización. Eran dispositivos dedicados encargados de crear las imágenes. En 1995 aparecen las primeras tarjetas 2D/3D con chips de gran potencia de cálculo. Desde entonces las mejoras se han orientado a obtener más potencia de cálculo y proceso de algoritmos 3D. Actualmente son plataformas programables y masivamente paralelas, y los propios fabricantes proporcionan lenguajes para trabajar sobre ellas.

La arquitectura de la GPU nos ofrece un pipeline que podemos observar en la Figura 3.1. Como se puede apreciar la GPU consta de múltiples cores. Éstos a su vez están formados por bloques de procesadores para operaciones en punto flotante, una caché de datos y una unidad de texturas cada uno. Se comunican a través de un bus con varias unidades Load/Store que permiten la transferencia de datos con la memoria global de la tarjeta. Consta además de un gestor de ejecución de hilos, que se encarga del reparto de trabajo. Es por tanto necesario que las aplicaciones exploten esta distribución para aprovechar al máximo los múltiples procesadores de los que se disponen. Denominamos warp al conjunto de dieciséis procesadores de cada chip, y half-warp a cada bloque de ocho procesadores de los que consta cada chip.
La clave de la arquitectura de la GPU reside en el uso de múltiples procesadores escalares sobre un flujo de datos. Se pueden agrupar por proximidad y en gran número para proporcionar una capacidad de cálculo muy potente. Poseen una lógica de decodificación y ejecución integrada de alta velocidad, y la memoria de cada chip puede ser leída muy rápidamente. Las instrucciones SIMD (single instruction, multiple data) se pueden implementar agrupando procesadores de manera eficiente, ya que las agrupaciones de procesadores paralelos masivos se adaptan muy bien a los cómputos de gráficos. Cada procesador escalar es completamente generalizado y desacoplado y soportan precisión de punto flotante de IEEE 754. El reloj que rige los procesadores escalares esta separado del reloj que rige el resto del chip. Si comparamos la arquitectura de 128 procesadores escalares, con otras existentes de 32 procesadores vectoriales de 4 componentes, los ingenieros de Nvidia han estimado que el incremento de rendimiento producido es de en torno a 2x.

3.1. Modelo de Programación

La llegada de las CPUs multicore y las GPUs manycore ha supuesto que los procesadores ahora puedan funcionar como sistemas paralelos.

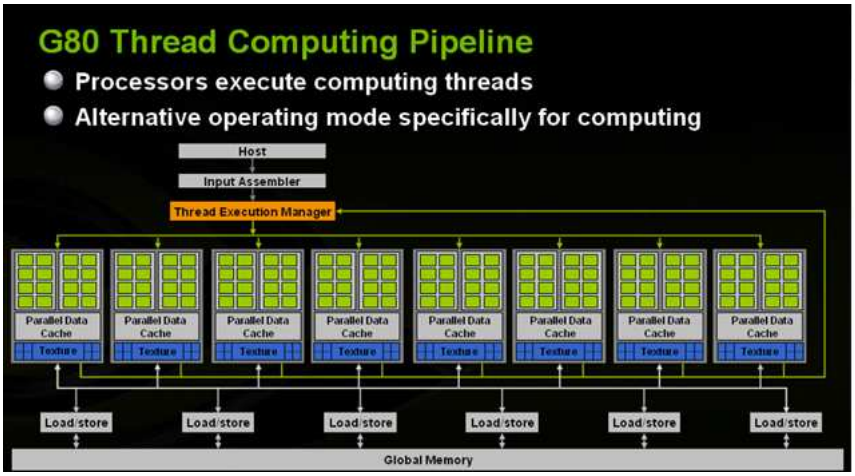


FIGURA 3.1: Pipeline de la GPU GeForce 8800

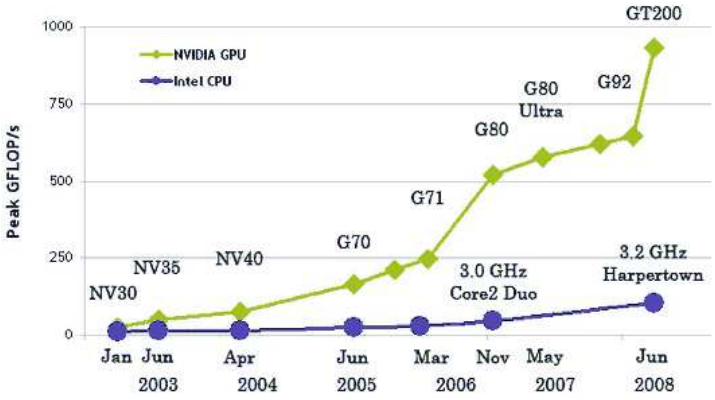


FIGURA 3.2: GFlops Nvida vs Intel

CUDA [] es un modelo de programación paralela y software diseñado para sobreponerse al reto de mantener una baja dificultad de aprendizaje para programadores familiarizados con el entorno C. En su base podemos encontrar 3 abstracciones: una jerarquía de grupos de hilos, memoria compartida y barreras de sincronización. Estas abstracciones proveen de un preciso paralelismo de datos, hilos y tareas. Guían al programador a dividir los problemas en particiones independientes sobre las que aplicar el paralelismo. Tal descomposición preserva la expresividad del lenguaje permitiendo que los hilos cooperen para solucionar cada uno de los subproblemas, al mismo tiempo que permite una escalabilidad transparente puesto que cada uno de los subproblemas se puede solucionar en cualquiera de las procesadores disponibles.
Llevados por las demandas del mercado del tiempo real, gráficos 3D de alta definición, la programación en GPU, las tarjetas han evolucionado hacia procesadores altamente paralelos, multihilo, manycore con tremendo poder computacional y muy alto ancho de banda de memoria. La razón tras esta discrepancia entre capacidad de punto flotante entre CPU

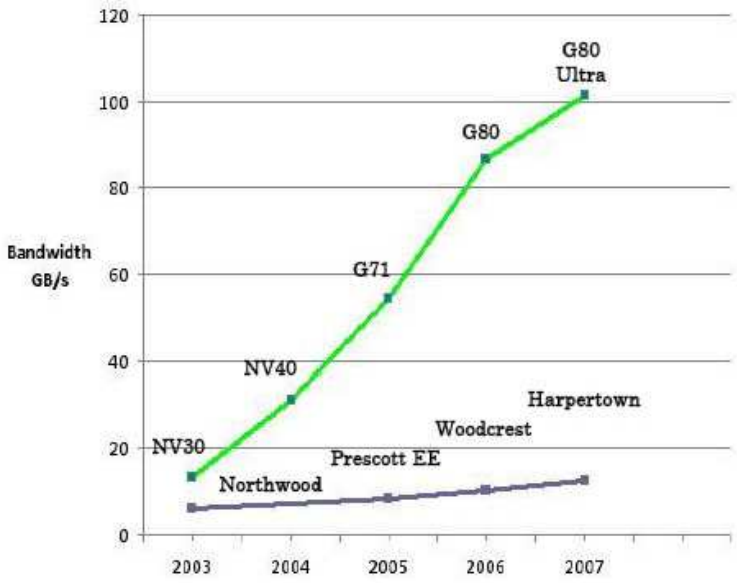


FIGURA 3.3: Comparativa ancho de banda GPU vs CPU

y GPU es que la GPU está especializada en cómputo intensivo, computación altamente paralela, exactamente lo que usa el renderizado de gráficos, y por lo tanto diseñada para tener más transistores destinados al procesamiento de datos más que a cache de datos y control del flujo. Más específicamente, una GPU es esencialmente eficiente para solucionar problemas que pueden expresarse como computación de datos paralelos, el mismo programa es ejecutado con muchos datos en paralelo, con mucha intensidad aritmética y hay mayor cantidad de operaciones aritméticas que operaciones de memoria.
Debido a que el mismo programa se ejecuta para cada elemento hay un requisito menor de control de flujo, y como es ejecutado en muchos elementos de datos y tiene gran intensidad aritmética la latencia de acceso a memoria queda escondida mientras se realizan otros cálculos, en lugar de hacer uso de una caché de datos.
Muchas aplicaciones que usan grandes conjuntos de datos pueden usar un modelo de progra-

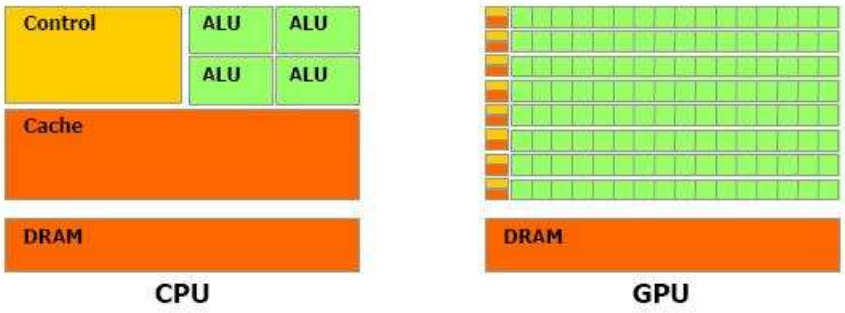


FIGURA 3.4: Estructura CPU y GPU

mación de datos paralelos para acelerar la computación. El renderizado de grandes conjuntos de píxeles de imágenes y vértices en 3D son asignados a hilos paralelos.
Similarmente, imágenes y procesamiento multimedia, que usan imágenes de vídeo, codificación o decodificación de vídeo, escalado de imágenes, etc, pueden asignar bloques de imágenes o píxeles a hilos y hacer que sean procesados paralelamente. De hecho, muchos algoritmos fuera del campo del renderizado y procesamiento de imágenes pueden ser acelerados usando procesamiento paralelo de datos.
El modelo de programación de CUDA está muy bien diseñado para mostrar las capacidad de paralelización de las GPUs.

3.1.1. Compute Unified Device Architecture

Compute Unified Device Architecture [NVIb, NVIc] (de aquí en adelante lo llamaremos por sus siglas, CUDA) extiende C permitiendo que el programador defina funciones en C, llamadas kernels, que son ejecutadas N veces en paralelo para N hilos diferentes, como contraposición a las funciones normales de C.
Los principales términos relacionados con los kernels son:

- Grid: es la división inicial que se aplica sobre el objeto. Cada Grid está formada por bloques. Por ejemplo, en una matriz de 16 por 16 podemos tener una Grid de 16 por 16 que abarca toda la matriz o definir 4 grid de 4 por 4.
- Bloques: conjuntos de hilos que componen los grid, se tratan de lanzar paralelamente. Normalmente el tamaño del bloque es mayor que el número real de hilos que se pueden ejecutar paralelamente, por lo que la GPU los ejecuta por warps o semiwarps.
- Warps: es el número de hilos máximo con el cual se puede trabajar en la GPU. Es común usar también semiwarps, los cuales son la mitad de hilos que un warp. Es habitual el uso del semiwarp dado que muchas veces el flujo de los hilos diverge y el warp saturaría la capacidad de procesamiento.
- Hilos: es la unidad de trabajo básica de los kernels. Cada hilo realiza trabajo sobre uno de los elementos del objeto inicial. Si el objeto inicial es una matriz tendremos que cada hilo trabaja sobre un elemento $a_{n,m}$ de la matriz.

Para obtener el máximo rendimiento a este modelo de programación debemos de tener en cuenta varios puntos:

- Maximizar paralelismo usando los recursos disponibles.
- Explotar la localidad de datos
- Uso eficiente de la memoria, alineando los accesos a los datos.

Un kernel es definido usando la declaración especifica _ _ global _ _ y el número de hilos CUDA para cada llamada se especifican usando una nueva sintaxis:

```
//Definición del kernel
__global__  void vecAdd(float* A, float* b, float* C){}

//Llamada al kernel
int main(){
      vecAdd<<<1, N>>>(A, B, C);
}
```

Cada uno de los hilos que se ejecutan por el kernel tiene una única ID de hilo que es accesible desde el kernel por una variable interna llamada.

3.1.1.1. Jerarquía de Hilos

Cada hilo puede ser identificado usando un índice unidimensional, bidimensional o tridimensional, formando un bloque de hilos unidimensionales, bidimensionales o tridimensionales según queramos representar el espacio del problema.
Esto provee una forma natural de llamar los elementos del dominio del vector, matriz o campo. Como un ejemplo, el siguiente código suma dos matrices A y B con tamaños NxN y guarda el resultado en la matriz C:

```
__global__ void matAdd(float A[N][N], float B[N][N], float C[N][N]){
      int i = threadIdx.x;
      int j = threadIdx.y;
      C[i][j]= A[i][j]+B[i][j];
}

int main(){

      //definimos el tamaño del bloque de hilos
      dim3 dimBlock(N,N);

      matAdd<<<1, dimBlock>>>(A, B, C);
}
```

Los hilos dentro de un bloque pueden cooperar entre ellos compartiendo datos a través de la memoria compartida y sincronizándose a través de la ejecución coordinada de accesos a memoria.
Para la eficiencia de la cooperación, se cuenta con que la memoria compartida es de una baja latencia cerca de cada núcleo de procesador y se espera que todos los hilos de un bloque residan en el mismo núcleo de procesador. El número de hilos por bloque está por lo tanto restringido por el recurso limitado de memoria de cada núcleo de procesador. En las últimas arquitecturas tesla de NVIDIA, un bloque de hilos puede contener hasta 512 hilos.
Cada bloque dentro de la rejilla se puede identificar usando un índice uni o bidimensional accesible dentro del kernel.
Los bloques de hilos son necesarios para ejecutar independientemente, debe ser posible ejecutarlos en cualquier orden, en paralelo o en serie. Este requisito de independencia permite que los bloques de hilos sean organizados de cualquier forma a lo largo del número de núcleos de que dispongamos, permitiendo escribir código escalable.
El número de bloques de hilos en una rejilla es típicamente dictado por el tamaño de los datos que van a ser procesados más que por el número de procesadores en el sistema, el cual puede exceder ampliamente.

3.1.1.2. Jerarquía de memoria

Los hilos en CUDA pueden acceder a los datos de múltiples espacios de memoria durante su ejecución. Cada hilo tiene una memoria local privada y cada bloque de hilos tiene una memoria compartida visible por todos los hilos del mismo bloque y en el mismo tiempo de ejecución del bloque. Finalmente, todos los hilos pueden acceder a la misma memoria global.
Hay dos espacios de memoria de sólo lectura adicionales accesibles por todos los hilos: los espacios de memoria constante y de texturas. Los espacios de memoria global, constante y de texturas están optimizadas para diferentes usos de memoria. La memoria de texturas además ofrece distinto tipo de direccionamiento, como filtro de datos, para algunos tipos de datos específicos. Los espacios de memoria global, constante y de texturas son persistentes a través de los lanzamientos de kernels de la misma aplicación.
Podemos considerar la memoria de la CPU como un nivel más externo de memoria. Las

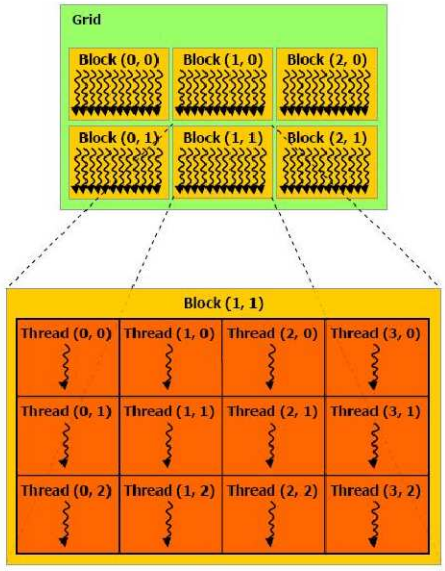


FIGURA 3.5: Rejilla de bloque de hilos

transferencias CPU-GPU son las más costosas, por lo que es importante minimizar estas transacciones. Además hemos de tener en cuenta que una vez que se lanza un kernel no es posible realizar estas comunicaciones.

3.1.1.3. Host y dispositivo

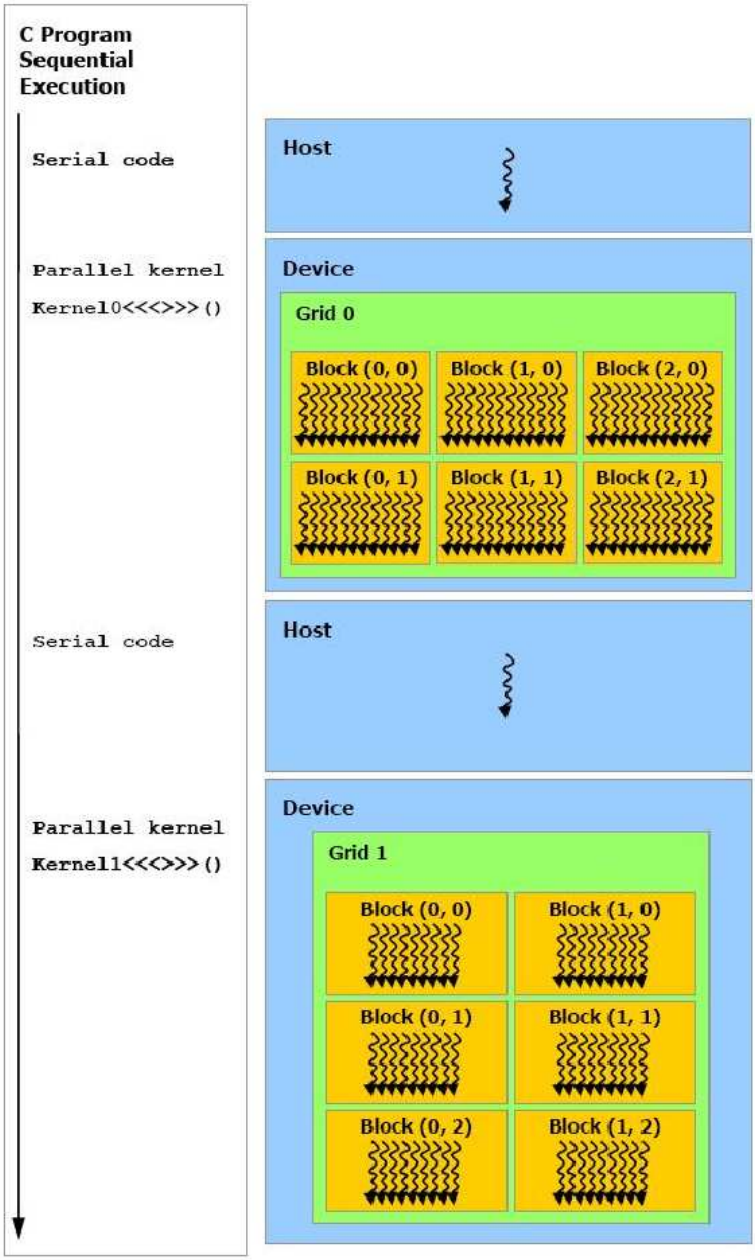
Como se ilustra en la Figura 3.1.1.3, CUDA asume que los hilos se pueden ejecutar en dispositivos separados físicamente que operan como coprocesador de un host corriendo un programa C. Este es el caso, por ejemplo, cuando los kernels se ejecutan en una GPU y el resto del programa C en la CPU.
CUDA también asume que tanto el host como el dispositivo mantienen su propia DRAM referida como memoria host y memoria del dispositivo, respectivamente. Por lo tanto, un programa maneja los espacios de memoria global, constante y de texturas visibles por los kernels a través de llamadas al entorno de CUDA. Esto incluye la asignación y liberación de la memoria del dispositivo, tanto como la transferencia de datos entre host y dispositivo.

3.1.1.4. Pila del software

La pila del software CUDA esta compuesta por diversas capas como se ilustra en la Figura 3.6: un driver de dispositivo, un interfaz de programación de aplicación (API) y su entorno (runtime), y dos librerías matemáticas de alto nivel de uso común, CUFFT y CUBLAS.

3.1.1.5. Capacidad de computación

La capacidad de computación del dispositivo esta definida por un número mayor de revisión y un número menor de revisión. Dispositivos con el mismo numero mayor de revisión



Serial code executes on the host while parallel code executes on the device.

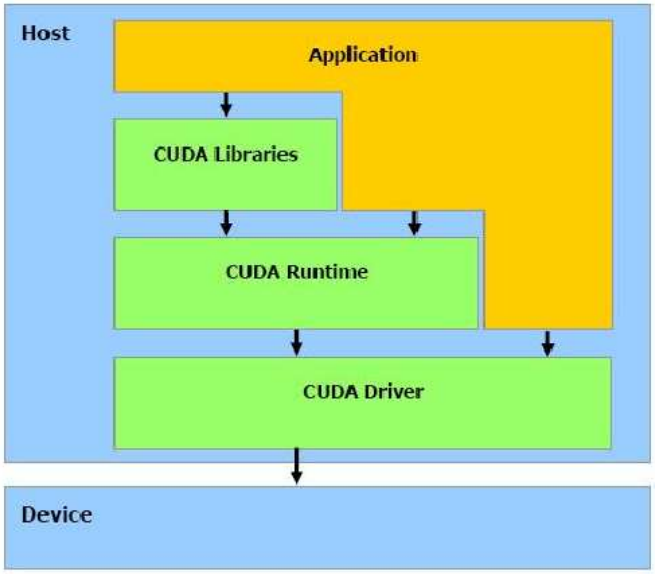


FIGURA 3.6: Arquitectura de la pila de software

tienen la misma arquitectura de base. El número de revisión menor corresponde a la mejora incremental a la arquitectura base, incluyendo posiblemente nuevas cualidades.

3.1.2. Implementación en la GPU

La arquitectura Tesla está construida alrededor de un array escalable de multiprocesadores de flujo multihilo(SMs). Cuando un programa CUDA en la CPU host invoca un rejilla de kernels, los bloques de la rejilla se enumeran y se distribuyen en multiprocesadores con capacidad de ejecución disponible. Los hilos de un bloque de hilos se ejecutan concurrentemente en un multiprocesador. Cuando los bloques de hilos terminan, nuevos bloques son lanzados en los multiprocesadores libres. Un multiprocesador consiste de ocho núcleos de procesador escalables (SP), dos unidades de función especial para transcendentales, una unidad de instrucción multihilo, y un chip de memoria compartida. Los multiprocesadores crean, manejan, y ejecutan hilos concurrentes en hardware sin ninguna planificación anterior.
Se implementa la barrera de sincronización intrínseca _ _ syncthreads() _ _ con una simple instrucción. proporciona sincronización rápida por barrera con baja carga de creación de hilos y ninguna planificación anterior, soportan eficientemente el paralelismo de grano fino, permitiendo, por ejemplo, una baja descomposición granular de los problemas asignando a cada elemento de un dato un hilo, como un píxel en una imagen o una celda en una computación basada en rejillas.
Para manejar cientos de hilos corriendo distintos programas, el multiprocesador emplea la arquitectura llamada SIMT (single-instruction, multiple-thread (instrucción simple, hilo múltiple)). El multiprocesador asigna cada hilo a un núcleo escalar de procesador, y cada escalar ejecuta un hilo independientemente con sus propias direcciones de instrucción y registro de estado. La unidad del multiprocesador SIMT crea, maneja, planifica, y ejecuta en grupos de 32 hilos paralelos llamados warps. Hilos individuales componiendo un SIMT warp empiezan en la misma dirección de programa a la vez pero son libres en todo lo demás de

bifurcarse y ejecutarse libremente.
Cuando un multiprocesador recibe uno o más bloques de hilos que ejecutar, los divide en warps que son planificados por la unidad del SIMT. El modo en que se divide un bloque en warps es siempre el mismo; cada warp contiene hilos consecutivos, incrementado las IDs de hilos con el primer warp conteniendo el Hilo 0.
Cada momento de lanzamiento de instrucciones, la unidad SIMT selecciona un warp que esté listo para ser ejecutado, lanza la siguiente instrucción del grupo de hilos del warp activo y planifica la siguiente instrucción. Un warp ejecuta una instrucción común a la vez, así que la eficiencia total se realiza cuando los 32 hilos de un warp convergen en el camino de ejecución. Si los hilos de un warp divergente debido a ramas dependientes de datos, el warp ejecuta en serie cada rama tomada, deshabilitando hilos que no están en esa rama, y cuando se completan todas las ramas, los hilos convergen de nuevo en el mismo camino de ejecución. Las divergencias de camino solo ocurren dentro de un warp; diferentes warps ejecutan independientemente sin importar si están ejecutando flujos de código comunes o distintos.
La arquitectura SIMT es semejante a la organización vectorial SIMD (single instruction, Multiple Data (instrucción simple, datos múltiples)) en una única instrucción que controla múltiples elementos a procesar. La diferencia clave es que la organización vectorial SIMD expone el ancho de SIMD al software, mientras que las instrucciones SIMT especifican el comportamiento de ejecución y ramificación de un único hilo. En contraste con máquinas vectoriales SIMD, SIMT permite a los programadores escribir código paralelo a nivel de hilo para hilos independientes, escalares, tanto como código de datos paralelos para hilos coordinados.
Con el propósito de la corrección, los programadores pueden esencialmente ignorar el comportamiento del SIMT, aunque se pueden dar mejoras substanciales cuando se cuida que los hilos de un warp no diverjan en su camino. En la práctica, esto es análogo al papel de las líneas de caché en el código tradicional, el tamaño de las líneas de caché pueden ser libremente ignoradas cuando se diseña por corrección pero deben ser consideradas en la estructura del código cuando se diseña para máxima eficiencia.
Las arquitecturas vectoriales, por otra parte, requieren que el software controle la carga entre vectores y su divergencia manualmente.
Como se ilustra en la Figura 3.7, cada multiprocesador tiene un chip integrado de memoria de uno de los cuatro siguientes tipos:

- Un set de registros de 32-bits por procesador.
- Una caché de datos paralela o una memoria compartida que es accedida por todos los núcleos escalares del procesador y que es donde residen los espacios de memoria compartida.
- Una caché constante de sólo lectura compartida por todos los núcleos escalares del procesador y que acelera las lecturas del espacio de memoria constante, la cual es una región de sólo lectura de la memoria del dispositivo.
- Una caché de texturas de sólo lectura que es compartida por todos los núcleos escalares del procesador y que acelera las lecturas del espacio de memoria de texturas, la cual es una región de solo lectura de la memoria del dispositivo, cada multiprocesador accede a la caché de texturas a través de la unidad de texturas que implementa varios modos de direccionamiento y de filtro de datos.

Los espacios de memoria local y global son regiones de lectura-escritura de la memoria del dispositivo y no están en caché. Cuantos bloques puede procesar un multiprocesador

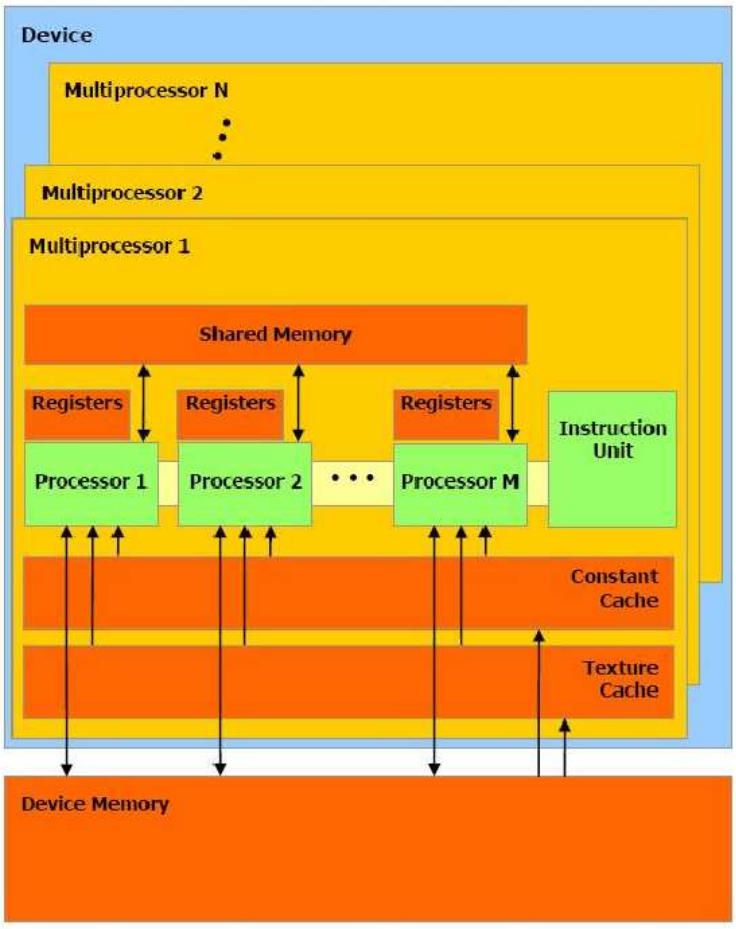


FIGURA 3.7: Un grupo de multiprocesadores Single Instruction Multiple Thread

a la vez depende de cuantos registros por hilo y cuanta memoria compartida por bloque son requeridas por un kernel, dado que los registros de un multiprocesador y su memoria compartida deben dividirse entre todos los hilos de la tanda de bloques. Si no hay suficientes registros o memoria compartida disponible por cada multiprocesador para procesar al menos un bloque, el kernel fallará al ser lanzado. Un multiprocesador puede ejecutar hasta ocho hilos de bloques de hilos concurrentemente.
Si una instrucción no atómica lanzada por un warp escribe en la misma dirección global o de memoria compartida para más de uno de los hilos de un warp, el número de escrituras en serie que ocurre en esa dirección y el orden en el que ocurre no está definido, pero al menos una de las escrituras se garantiza que ocurrirá. Si una instrucción atómica ejecutada por un warp trata de leer, modificar, o escribir en la misma dirección de memoria global para más de uno de los hilos del warp, cada lectura, modificación, o escritura a esa dirección ocurrirá, y serán en serie, pero el orden en que ocurrirán no está definido.

3.1.2.1. Múltiples dispositivos

Se garantiza el funcionamiento del uso de múltiples GPUs en dispositivos CUDA por una misma aplicación corriendo en un sistema multi-GPU solamente si estas GPUs son del mismo tipo. Sin embargo, si el sistema esta en modo SLI, solo una GPU puede ser usada como un dispositivo CUDA dado que todas las GPU's están unidas a bajo nivel en la pila del driver. El modo SLI necesita ser apagado en el panel de control de CUDA para ser capaces de ver cada una de las GPUs como dispositivos independientes.

3.1.2.2. Cambio de modo

Las GPUs dedican algo de la memoria DRAM a la llamada superficie primaria, la cual es usada como refresco del display del dispositivo cuando una salida es vista por el usuario. Cuando un usuario cambiando la resolución o profundidad de bits inicia un cambio de modo del display, la cantidad de memoria que necesita la superficie primaria cambia. Si un cambio de modo incrementa la memoria necesitada por la superficie primaria, el sistema puede tener que consumir la memoria reservada para las aplicaciones de CUDA, resultando en un error de estas aplicaciones.

3.1.3. Componente runtime del Host

Varios hilos del host pueden ejecutar código del dispositivo en el mismo dispositivo, pero por diseño, un hilo de un host solo puede ejecutar código en un dispositivo. Como consecuencia, múltiples hilos del host son requeridos para ejecutar código de dispositivo en diversos dispositivos.

3.1.3.1. Memoria

La memoria puede ser asignada como memoria lineal o como arrays de CUDA.
La memoria lineal existe en el dispositivo en una espacio de direcciones de 32 bits, por lo que entidades separadas en asignación pueden referenciarse una a otra por medio de punteros, por ejemplo, en un árbol binario. Los arrays CUDA son diseños opacos de memoria optimizados para traer texturas. Son unidimensionales, bidimensionales o tridimensionales y están compuestos de elementos, cada uno tiene 1,2 o 4 componentes que pueden tener enteros de 8,16 o 32 bits con o sin signo, o floats de 16 o 32 bits. Los arrays CUDA solo pueden ser leídos por kernels. Tanto la memoria lineal como los arrays CUDA se pueden leer o escriben por el host por medio de las funciones de copia de memoria descritas después. El runtime

del host también provee de funciones para asignar y liberar paginas, memoria bloqueada del host, en contraposición a la memoria normal paginable asignada por malloc(). Una ventaja de esta páginas de memoria bloqueadas es que el ancho de banda entre memoria del host y del dispositivo es mayor si la memoria se asigna de esa forma, solo para transferencias de datos realizadas por hilos del host que asignan memoria del host. Sin embargo, esta memoria es un recurso escaso, así que la asignación de esta forma empezará a fallar mucho antes de la asignación en memoria paginable. Además, reduce la cantidad de memoria física disponible para la operación del sistema para paginación, asignar mucha memoria de páginas bloqueadas reduce el rendimiento general del sistema.

3.1.3.2. Runtime API

Inicialización No hay una función explicita de inicialización para el API runtime. Inicializa la primera vez que una función del runtime es llamada.

Administración del dispositivo cudaGetDeviceCount() y cudaGetDeviceProperties() proveen de una forma de enumerar estos dispositivos y adquirir sus propiedades, cudaSetDevice() se usa para seleccionar el dispositivo asociado al hilo del host:

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
      cudaDeviceProp deviceProp;
      cudaGetDeviceProperties(&deviceProp, device);
}
cudaSetDevice(device);
```

Un dispositivo debe seleccionarse antes de una función __ global__ o cualquier función del API runtime sea llamada. Si esto no se hace con una llamada explicita a cudasetDevice(), el dispositivo 0 es seleccionado automáticamente y cualquier otra llamada siguiente a cudaSetDevice() no tendrá efecto.

Administración de Memoria La memoria local es asignada usando cudaMalloc() o cudaMallocPitch() y liberada usando cudaFree(). Los siguientes cogidos de ejemplo asignan un array de 256 elementos de punto flotante en memoria lineal:

```
float* devPtr;
cudaMalloc((void**)&devPtr, 256 * sizeof(float));
```

cudaMallocPitch() se recomienda para asignaciones de arrays de dos dimensiones ya que se asegura de que la reserva se realiza de forma adecuada para satisfacer los requisitos de alineamiento. Los arrays CUDA se asignan usando cudaMallocArray() y se liberan usando cudaFreeArray(). cudaGetSymbolAddress() se usa para recoger la dirección apuntando a la memoria reservada usando una variable declarada en el espacio global de memoria.

3.1.4. Rendimiento

3.1.4.1. Instrucciones matemáticas

Para lanzar una instrucciones para un warp, un multiprocesador necesita:

- 4 ciclos de reloj para:
  - Suma, multiplicación o suma múltiple de un punto flotante de precisión simple.
  - Suma de enteros.

- Operación sobre un bit, instrucciones de conversión de tipo.

- 16 ciclos de reloj para:

  - Elementos opuestos, raíces cuadradas opuestas,__ logf(x)
  - Multiplicaciones de enteros de 32 bit toman 16 ciclos de reloj, pero mul24 y _ umul24 dan una multiplicación de enteros con o sin signo de 24 bits en 4 ciclos de reloj.
  - Divisiones de enteros y operaciones de módulo son especialmente costosas y deben ser evitadas si es posible o reemplazadas por operaciones sobre bit siempre que se pueda.

3.1.4.2. Instrucciones de control de flujo

Cualquier instrucción de control de flujo (if, switch, do, for, while) pueden impactar significativamente en el rendimiento de los hilos al causar que hilos del mismo warp diverjan. Si esto ocurre, los diferentes caminos de ejecución necesitan ser serializados, incrementando el número total de instrucciones ejecutadas para el warp. Cuando se ejecutan todas las divergencias, los hilos convergen en el mismo camino de ejecución.
Para obtener un mejor rendimiento en algunos casos donde el control de flujo depende de las ID de los hilos, la condición de control debería estar escrita para minimizar el número de hilos divergentes. Esto es posible debido a que la distribución de los warps a lo largo del bloque es determinista. Algunas veces el compilador puede desenrollar bucles del if usando o un predicado de ramificación o la directiva # pragma unroll.

3.1.4.3. Instrucciones de Memoria

Las instrucciones de memoria incluyen cualquier instrucción que lea de o escriba a memoria compartida o global. Los accesos a memoria local solo ocurren para algunas variables automáticas.
Un multiprocesador necesita 4 ciclos de reloj para lanzar una instrucción de memoria para un warp. Cuando se accede a memoria local o global, hay además, 400 a 600 ciclos de reloj de latencia de memoria. Como un ejemplo, la operación de asignación del siguiente código:

```
__shared__  float shared[32];
__device__  float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

toma 4 ciclos de reloj para lanzar la lectura de memoria global, 4 ciclos de reloj para lanzar la escritura a memoria compartida, pero sobre todo toma 400 a 600 ciclos de reloj para leer un float de memoria global. Mucha de esta latencia de memoria global se puede esconder tras el planificador de hilos si hay suficientes operaciones aritméticas independientes para ser lanzadas mientras se espera que se complete el acceso a memoria global.

3.1.4.4. Instrucciones de sincronización

__ syncthreads__ toma 4 ciclos de reloj para lanzar para un warp si no hay hilos que tengan que esperar a otros hilos.

3.1.4.5. Ancho de banda de Memoria

El tamaño del ancho de banda efectivo para cada espacio de memoria depende significativamente del modelo de acceso a memoria usado. Dado que la memoria del dispositivo tiene mucha mayor latencia y menor ancho de banda que la memoria integrada, el acceso a memoria del dispositivo debería ser minimizado. Un típico modelo de programación es organizar los datos que llegan de la memoria del dispositivo en la memoria compartida, en otras palabras, para que cada hilo de un bloque tenga disponible:

- Cargar datos de la memoria del dispositivo en la memoria compartida.
- Sincronizar con todos los otros hilos del bloque de forma que cada hilo puede leer con seguridad la memoria compartida de la localización donde se escribió por distintos hilos.
- Procesar datos en memoria compartida.
- Sincronizar de nuevo si es necesario para asegurarse de que la memoria compartida se ha actualizado con el resultado.
- Escribir el resultado en la memoria del dispositivo.

Memoria Global El espacio de memoria global no tiene caché, por lo que es más importante aun el modelo de acceso correcto para maximizar el uso de ancho de banda, espacialmente dado lo costosos que son los accesos a la memoria del dispositivo. El dispositivo es capaz de leer palabras de 32,64 o 128 bits de memoria global a registros con una única instrucción. Para tener asignaciones como:

```
__device__ type device[32];
type data = device[tid];
```

compiladas en una única instrucción, el tipo debe ser tal que sizeof(tipo) sea igual a 4,8 o 16 y que la variable de tipo esté alineada con los bytes de sizeof(tipo). Para estructuras, los requisitos de tamaño y alineamiento se pueden forzar por el compilador usando los especificadores de alineamiento __ align__ (8) o __ align__ (16). Para estructuras mayores de 16 bytes, el compilador necesita generalmente cargar varias instrucciones. Para asegurarse que eso toma el mínimo numero de instrucciones tales estructuras deben ser definidas como __ align__ (16). Cualquier dirección de una variable residente en la memoria global o devuelta por una de las rutinas de asignación de memoria del driver o el API runtime esta siempre alineado al menos a 256 bytes. El ancho de banda de memoria global es usado más eficientemente cuando múltiples accesos simultáneos a memoria por hilos en un semi-warp pueden ser fusionados en una sola instrucción de transacción. El tamaño de una instrucción de transacción puede ser de 32, 64 o 128 bytes.

Memoria Local Los accesos a memoria local solo ocurren para algunas variables automáticas. Como el espacio de memoria global, el espacio de memoria local no tiene caché, por lo que sus accesos son igual de costosos que a memoria global. Estos accesos son siempre fusionados dados que son por definición accesos por hilo.

Memoria Constante El espacio de memoria constante tiene caché por lo que una lectura de memoria constante solo cuesta una lectura de memoria de dispositivo en un fallo de lectura de la caché. Para todos los hilos de un semi-warp, leer de la caché constante es tan rápido como leer de un registro, siempre que todos los hilos lean de la misma dirección. El coste aumenta linealmente con el numero de hilos leyendo de diferentes direcciones.

Memoria Compartida Como está integrada en el chip, el espacio de memoria compartida es mucho más rápido que el espacio de memoria local o global. De hecho, para todos los hilos de un warp, acceder a memoria compartida es tan rápido como acceder a un registro siempre que no haya conflictos de bancos entre hilos. Para conseguir un alto ancho de banda de memoria, la memoria compartida esta dividida en módulos de memoria de tamaño igual, llamados bancos, que se acceden simultáneamente. Así que, cualquier lectura o escritura de memoria requerida para n direcciones que caen en n bancos de memoria distintos el servicio puede ser hecho simultáneamente, dando un ancho de banda efectivo de n veces el ancho de banda de un solo modulo. Sin embargo, si dos peticiones de memoria caen sobre el mismo banco de memoria, existe un conflicto de bancos y el acceso tiene que ser serializado. El hardware divide las peticiones a memoria con conflictos de banco en tantas peticiones libres de conflicto separadas como sea necesario, decrementando el ancho de banda efectivo por un factor igual al numero de separado de peticiones de memoria. Si el numero de peticiones inicial es n, la petición inicial de memoria se dice que causa conflictos de banco de n direcciones. Para tener el máximo rendimiento, es por lo tanto importante entender como la memoria asigna los bancos de memoria para planificar las peticiones de forma que causen el mínimo número de conflictos de bancos. En el caso del espacio de memoria compartida, los bancos se organizan de forma que sucesivas palabras de 32 bits se asignan a bancos sucesivos y cada banco tiene un ancho de banda de 32 bits cada 2 ciclos de reloj. Una caso común de acceso a una palabra de 32 bits de un array indexado por el ID del hilo y una desplazamiento s:

```
shared__  float shared[32];
float data = shared[BaseIndex + s * tid];
```

En este caso, los tid del hilo y el tid+n accede al mismo banco de memoria cuando s*n es un múltiplo de m/d donde d es el máximo común divisor entre m y s. Como consecuencia, no habrá solo conflicto de bancos si la mitad del tamaño de warp es menor o igual que m/d. La Figura 3.8 muestran algunos casos de accesos a memoria sin conflictos mientras que la 3.9 muestra ejemplos de accesos que causan conflicto de bancos. Otros casos dignos de ser mencionados son aquellos en que cada hilo accede a un elemento que es más pequeño o mayor que 32 bits. Por ejemplo, hay conflictos de banco si a un array de char se le accede de la siguiente forma:

```
shared__ char shared[32];
char data = shared[BaseIndex + tid];
```

porque shared[0], shared[1], shared[2], y shared[3], por ejemplo, pertenecen al mismo banco. No hay conflictos sin embargo, si al mismo array se le accede de la siguiente forma:

```
char data = shared[BaseIndex + 4 * tid];
```

También hay errores de conflicto de banco de 2 direcciones para arrays de double:

```
shared__ double shared[32];
double data = shared[BaseIndex + tid];
```

Dado que la petición de memoria se compila en dos peticiones separadas de 32 bits. Una asignación de estructura se compila siempre en tantas peticiones de memoria como sea necesario para cada miembro de la estructura, en el siguiente código, por ejemplo:

```
__shared__ struct type shared[32];
struct type data = shared[BaseIndex + tid];
```

resulta en:

- Tres lecturas separadas de memoria sin conflicto de banco si el tipo esta definido como

```
struct type {
float x, y, z;
};
```

dado que cada miembro es accedido con un desplazamiento de tres palabras de 32 bits.

- Dos lecturas separadas de memoria con conflictos si tipo se define como

```
struct type {
float x, y;
};
```

dado que cada miembro es accedido con n desplazamiento de dos palabras de 32 bits.

- Dos lecturas separadas de memoria con conflictos de memoria si el tipo se define como

```
struct type {
float f;
char c;
};
```

dado que cada miembro es accedido con un desplazamiento de cinco bytes.

Finalmente, la memoria compartida también ofrece un mecanismo de transmisión (broadcast) donde una palabra de 32 bits se puede leer y transmitir a varios hilos simultáneamente cuando se sirve una sola petición de lectura de memoria. Esto reduce el número de conflicto de banco cuando muchos hilos de un semi-warp leen de una dirección con la misma palabra de 32 bits. Más precisamente, una petición de lectura de memoria hecha por muchas direcciones se sirve en muchas pasos en el tiempo, un paso cada dos ciclos de reloj, dando un subconjunto de servicios libres de conflicto de estas direcciones por paso hasta que todas las direcciones son servidas. A cada paso, el subconjunto se construye con las sobrantes direcciones de memoria que todavía no se han dado usando el siguiente proceso:

- Seleccionar una de las palabras apuntadas por las sobrantes direcciones de memoria como la palabra a transmitir.
- Incluir en el subconjunto:
  - Todas las direcciones que están dentro de la palabra a emitir.
  - Una dirección para cada banco apuntando a el resto de las direcciones.

Que palabra se elige como la palabra a emitir y cual es la dirección seleccionada para cada banco en cada ciclo no esta especificado. Un caso común libre de conflictos es cuando todos los hilos de un semi-warp leen de una dirección dentro de la misma palabra de 32 bits.

Registros Generalmente, acceder a los registros no significa ningún ciclo de reloj por instrucción, pero retrasos pueden ocurrir debido a dependencias LDE de registro y conflictos de banco de memoria de registros. Este retraso introducido por dependencias LDE se puede ignorar tan pronto como haya al menos 192 hilos activos por multiprocesador para esconderlos. El compilador y el planificador de hilos planifican las instrucciones de forma óptima para evitar tantos conflictos de banco de memoria de registros como sea posible. Se obtienen mejores resultados cuando el número de hilos por bloque es múltiplo de 64.

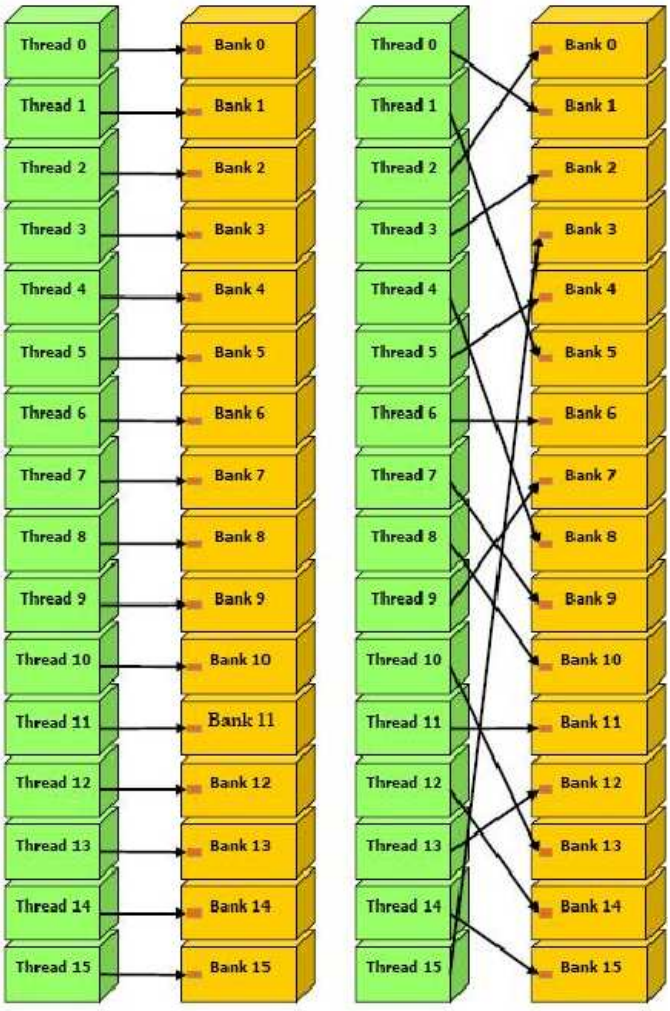


FIGURA 3.8: Accesos alineados con salto de 1 palabra de 32 bits, o con permutación aleatoria

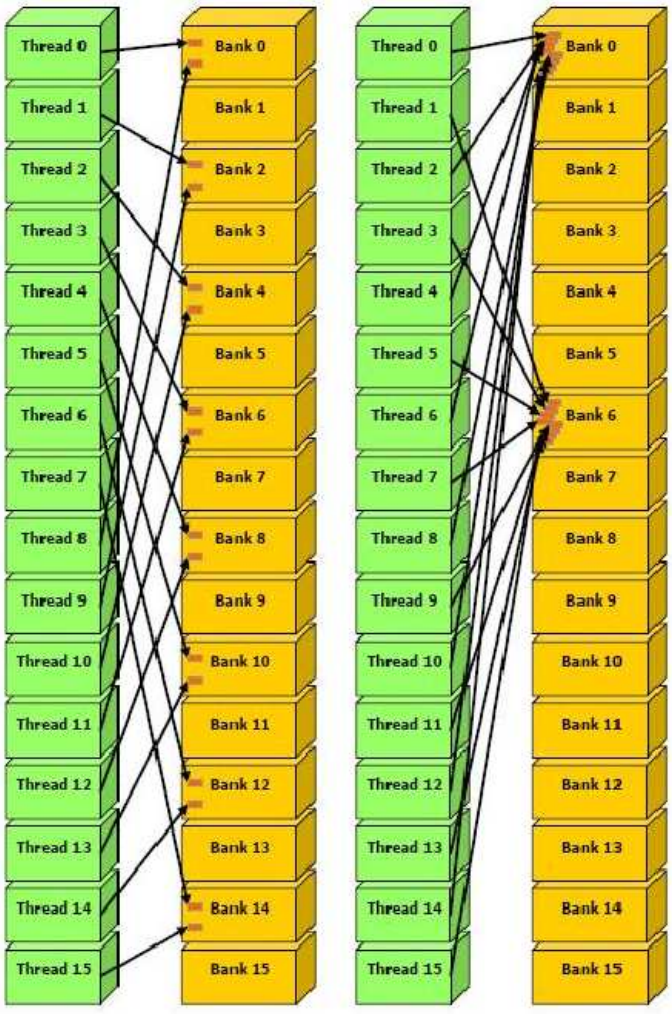


FIGURA 3.9: Accesos alineados con conflictos

3.1.4.6. Hilos por Bloque

Dados un número total de hilos por rejilla, el número de hilos por bloque o el equivalentemente el número de bloques, estos deben ser elegidos para maximizar la utilización de los recursos de computación. Esto significa que al menos debe haber tanto bloques como multiprocesadores en el dispositivo. Aun más, correr solo un bloque por procesador fuerza el multiprocesador a esperar durante sincronización de hilos y también durante lecturas de la memoria del dispositivo si no hay suficientes hilos por bloque para ocultar la latencia de carga. Por lo tanto suele ser mejor permitir que dos o más bloques estén activos en cada multiprocesador permitiendo el solapamiento de al menos el doble de tantos bloques como multiprocesadores en el dispositivo, pero también la cantidad de memoria compartida asignada por bloque debe ser al menos la mitad del total de la memoria compartida disponible por procesador. Un flujo mayor de bloques de hilos de en tubería de esta forma amortizan aun más. Con suficiente número de bloques, el número de hilos por bloque debe ser elegido como un múltiplo del tamaño del warp para evitar el gasto de recursos de computación con warp no llenos, o mejor, un múltiplo de 64 como se explicó anteriormente. Asignar más hilos por bloque es mejor para una partición efectiva del tiempo, pero más hilos por bloque significan menos registros por hilo. Esto puede causar el fallo de la llamada al kernel si este compila en más registros que los permitidos por la configuración de la ejecución. 64 hilos por bloque es el mínimo y solo tiene sentido si hay múltiples bloques activos por procesador. 196 o 256 son mejor y usualmente permite suficientes registros para compilar.

3.1.4.7. Transferencias entre Host y Dispositivo

El ancho de banda entre dispositivo y la memoria del dispositivo es mucho mayor que el que hay entre la memoria del dispositivo y la memoria del host. Por lo tanto, deben minimizarse las transferencias entre host y dispositivo, por ejemplo, moviendo, más código del host al dispositivo, aunque eso signifique correr kernels con menos paralelismo. Estructuras de datos intermedias pueden ser creadas en la memoria del dispositivo, operadas en el dispositivo y destruidas allí sin ser asignadas por el host o copiadas a la memoria del host.

Capítulo 4

Implementación de Aplicaciones en GPU

En este capítulo vamos a presentar los métodos que hemos elegido implementar puesto que son los que hemos considerado más representativos y fiables en la identificación de imágenes.

Para implementar un programa en GPU en primer lugar debemos determinar los núcleos paralelos, por lo que para cada algoritmo presentamos los kernels que hemos identificado y como hemos distribuido los datos para realizar los cálculos.

Posteriormente mostramos los kernels más representativos y las mejoras que hemos realizados sobre los mismos.

Finalmente mostramos los resultados obtenidos para cada método.

4.1. Factorización no negativa de matrices NMF

La implementación del algoritmo NMF requiere tener disponibles los datos de la matriz original y de las matrices auxiliares que se generan durante el proceso.

En las Figuras 4.1 podemos ver una traza de los cálculos que realiza el algoritmo NMF original. En los recuadros se muestran los kernels que hemos identificado como más importantes para el método, como la multiplicación o reducción de matrices.
Puesto que en nuestro caso trabajamos sobre una tarjeta gráfica, el tamaño de la memoria de la misma será un factor determinante a la hora de disponer de los datos de dichas matrices.

Partimos del hecho de que para realizar el entrenamiento será necesario trabajar con matrices que ocupen 500 Mbytes de memoria global en la tarjeta, y esto sin incluir el resto de matrices auxiliares que también requieren espacio en la memoria de la tarjeta.

Es necesario por tanto partir los datos y realizar los cálculos parciales en la tarjeta, para en una serie de pasos realizar el calculo completo de W y H.

El número de elementos en los que debemos dividir la matriz original depende tanto del tamaño de la misma como de la memoria del dispositivo. Si la memoria del dispositivo es suficiente para albergar a la matriz original y las auxiliares no será necesario dividirla, y si es insuficiente deberemos crear trozos de la matriz original de modo que quepan en memoria de la tarjeta una parte de la matriz original y todas las matrices auxiliares requeridas.

El algoritmo NMF realiza varias multiplicaciones y reducciones de matrices, y puesto que son unas operaciones con un gran coste computacional decidimos dedicar un mayor

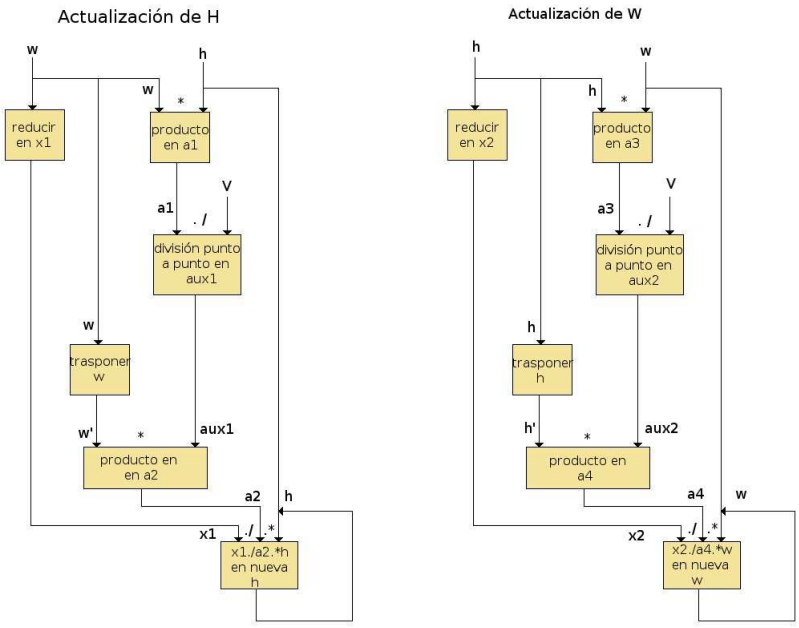


FIGURA 4.1: Traza del algoritmo NMF

esfuerzo a tratarlas. Posteriormente también se usan en la parte que hemos tratado del segundo algoritmos de reconocimiento.

A continuación mostramos los kernel a través de los cuales hemos experimentado con diferentes opciones y los resultados que obtuvimos.

4.1.1. Multiplicación de matrices

Inicialmente paralelizamos el código básico de multiplicación de matrices A y B, que consiste en, para cada posición (i,j) de la matriz resultado realizar la suma del producto de los elementos de la fila i-ésima de la matriz A por los elementos de la columna j-ésima de la matriz B, como podemos ver en la Figura 4.2.

4.1.1.0.1. Multiplicación en Memoria Global El primer paso es crear una rejilla sobre la matriz resultado y establecer el tamaño de cada bloque. Definimos bloques cuadrados de 16x16 hilos, por lo que dispondremos de 256 hilos en total por bloque, suficientes para llenar un Warp completo y aprovechar al máximo los procesadores, y creamos la rejilla sobre la matriz resultado. Ahora copiamos las matrices A y B en memoria global de la tarjeta, y reservamos espacio para la matriz resultado.

Lanzamos el kernel donde cada hilo de un bloque de hilos se encargará de utilizar la fila y columna que necesita de las matrices A y B en memoria global y operará con ellos hasta obtener el valor de su posición. Cuando termina el kernel, recuperamos la matriz resultado desde la tarjeta, y liberamos la memoria.

En el Cuadro 4.1 podemos ver los resultados de pruebas con diferentes tamaños.

4.1.1.0.2. Multiplicación en Memoria Compartida Podemos mejorar el kernel anterior si hacemos uso de la memoria compartida de la tarjeta, puesto que el acceso a memoria

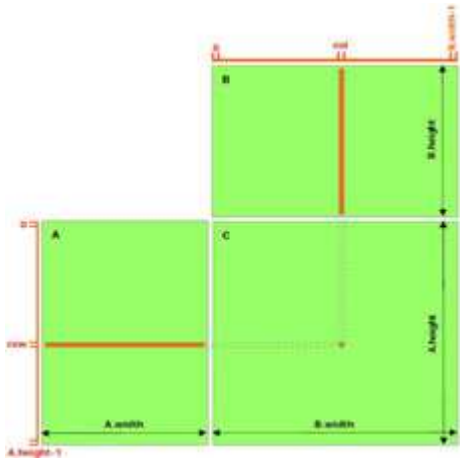


FIGURA 4.2: Multiplicación de matrices en memoria global

Caso	Filas	Columnas	Tiempo(ms)
1	256	256	39.11
2	512	512	343.41
3	1024	1024	3356.06
4	2048	2048	53064.51

TABLA 4.1: Resultados en Memoria Global con bloque de 16x16

global es muy costoso, en torno a 400 ciclos.

En esta segunda versión cada bloque de hilos reserva memoria compartida para las filas y columnas que requerirán. Además los hilos se encargarán de traer desde memoria global a compartida los datos de forma que el acceso a estos sea alineado para que sea más eficiente. Para acceder de forma alineada haremos que los hilos cooperen para traer a memoria compartida los datos que requiere el bloque completo de hilos, logrando así que no haya competencia por los datos, ni que haya múltiples lecturas del mismo dato a memoria global.

En el kernel anterior necesitábamos reservar memoria compartida para 16 filas y 16 columnas, lo cual, si la matriz es muy grande, no siempre es factible. La siguiente mejora que introducimos es que cada bloque de hilos traiga progresivamente los datos de las filas y columnas. En el kernel haremos que se reserve espacio en memoria compartida para dos bloques de 16x16 de datos de las matrices A y B, e iteraremos sobre las filas/columnas añadiendo el resultado parcial en cada iteración hasta tener el resultado final.

En la figura 4.3 podemos ver una gráfica del proceso. En el Cuadro 4.2 podemos ver los resultados de utilizar la memoria compartida de modo bloqueado, y podemos observar que cuanto mayor es el tamaño de la matriz mayor rendimiento obtenemos con respecto a usar memoria global directamente, en particular, para el caso de 2048x2048 el tiempo de proceso es 125 veces menor. Podemos ver una comparación del tiempo empleado en la multiplicación en la Figura 4.4

4.1.1.0.3. Tratamiento de bordes Puesto que las matrices A y B pueden ser de cualquier tamaño y el bloque que hemos elegido es de tamaño fijo 16x16 cuando nos encontremos

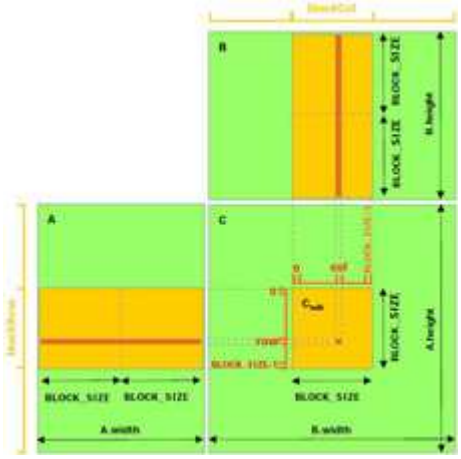


FIGURA 4.3: Cálculo de bloques por proceso iterativo

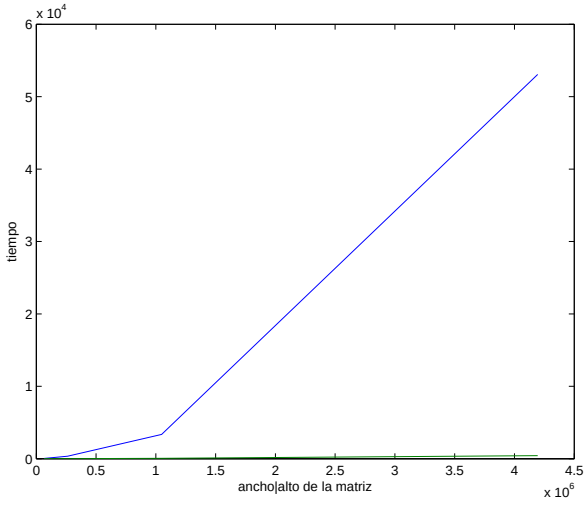


FIGURA 4.4: Tiempo para memoria global y compartida

Caso	Filas	Columnas	Tiempo(ms)
1	256	256	1.07
2	512	512	7.01
3	1024	1024	49.33
4	2048	2048	424.47

TABLA 4.2: Resultados en Memoria Compartida, con acceso por bloques, y tamaño de bloque 16x16

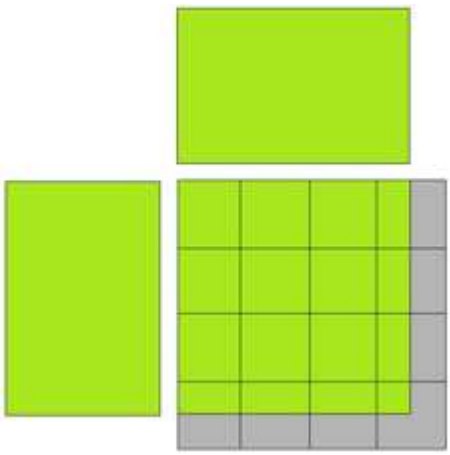


FIGURA 4.5: Tratamiento de bordes

con que las dimensiones de A y B no sean múltiplo de 16, tendremos que tratar los bloques que se encuentran en los bordes. Nos enfrentamos a dos tipos de problema, el primero, cuando las filas de A o las columnas de B no sean múltiplo de 16, la matriz resultado tampoco lo será en la dimensión correspondiente. Para tratarlo añadimos en el código del kernel una bifurcación de modo que, si nos encontramos en este caso, los hilos que se encuentran en la zona comprometida, no tendrán que operar, sin embargo deben seguir cooperando con los demás hilos para traer los datos de A y B.
Otro caso a tratar aparece cuando las columnas de A y por tanto las filas de B no son múltiplo de 16, aquí el problema son las lecturas a memoria compartida iterativas sobre A y B, ya que en la última iteración no todos los hilos han de leer. Para solventar esta situación añadimos la condición precisa en el código de lectura a memoria compartida del kernel para impedir accesos ilegales.

Las instrucciones de bifurcación dentro del código son bastante ineficientes, salvo que para cada bloque todos los hilos tomen el mismo camino, en cuyo caso el rendimiento no se ve afectado tan drásticamente.

En nuestro código sólo los bloques del borde derecho e inferior de la rejilla tienen que tomar la bifurcación, y el resto de bloques, la gran mayoría ejecuta la misma rama del bloque, y dado que trabajamos con matrices de gran tamaño, una baja proporción de los bloques ejecutan la parte de tratar bordes. En concreto para una matriz de 15000x15000 necesitaremos una rejilla de 938x938, donde 877969 bloques no tomaran la rama de tratamiento de bordes, y 1875 tendrán hilos que irán por una u otra rama, 1 de cada 470 bloques.

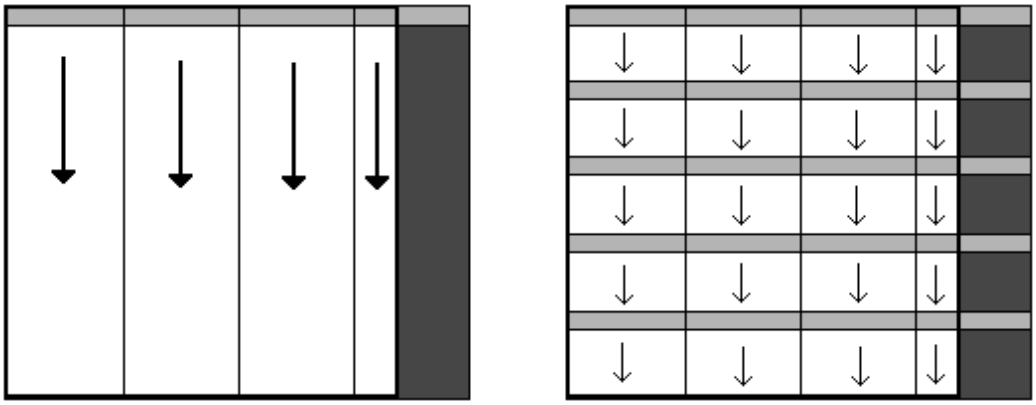


FIGURA 4.6: Dos opciones de rejilla para la reducción de una matriz

4.1.2. Reducción de una matriz

Este kernel se encarga de la reducción de una matriz a una fila o columna, sumando todos los elementos de cada columna o fila respectivamente.

La primera decisión que hay que tomar es como definir la rejilla, las dos opciones que tenemos son:

- Una rejilla unidimensional, en donde cada bloque trata un número de filas, o columnas, completas.

- Una rejilla bidimensional, donde cada grupo de filas, o columnas, se trata por K bloques. Con esta elección tendremos que hacer un proceso iterativo donde en cada iteración reducimos la matriz en un factor K hasta que finalmente tengamos solamente una fila o columna, se trata por tanto de una reducción logarítmica.

En la Figura 4.6 podemos observar las dos opciones. La eficiencia de una u otra elección de rejilla dependerá de como sea la matriz. Si disponemos de un bloque unidimensional de 128 hilos, y la matriz no tiene tamaño suficiente para que trabajen todos, en el caso de la rejilla unidimensional solo crearemos un bloque que tendrá que hacer todo el trabajo, y por lo tanto estaremos desperdiciando la potencia de cálculo de la tarjeta. Creando una rejilla bidimensional, tendremos k bloques que se encargarán de realizar el trabajo de forma cooperativa.

En caso de que la matriz sea de tamaño suficientemente grande, las dos opciones habrán creado suficientes bloques de hilos como para que se aproveche enteramente la capacidad de la tarjeta, aunque hay que tener en cuenta que la opción logarítmica debe lanzar más kernels, hasta haber reducido la matriz por completo, por lo que estará penalizada por este hecho.

En el Cuadro 4.3 podemos ver una tabla con tiempos para diferentes tamaños de matriz, comparando las dos opciones. podemos observar que la reducción logarítmica es más eficiente mientras que la dimensión sobre la que reducimos es menor o igual que el tamaño de bloque. Para tamaños mayores es mejor lanzar el kernel normal, puesto que puede aprovechar el uso de la tarjeta con sólo lanzar un kernel.

Tamaño	Normal	Logarítmica
100x20	0.060	0.049
1000x20	0.396	0.125
4000x20	1.518	0.275
8000x20	3.009	0.480
100x100	0.061	0.126
1000x100	0.402	0.342
4000x100	1.542	1.011
8000x100	3.064	2.144
100x500	0.062	0.203
1000x500	0.409	1.279
4000x500	1.554	5.864
8000x500	3.072	10.803
100x1000	0.063	0.311
1000x1000	0.412	2.778
4000x1000	1.550	10.882
8000x1000	3.059	20.396
100x8000	0.098	2.197
1000x8000	0.462	20.202
4000x8000	1.650	77.129
8000x8000	3.243	152.121

TABLA 4.3: Tiempos (ms) de las reducciones con tamaño de bloque 128

4.1.3. Desarrollo del programa

El programa desarrollado para el calcula W y H a través de un proceso iterativo acotado por el usuario. Inicialmente damos valores aleatorios a las matrices y a partir de ese momento vamos actualizando en cada iteración los valores del paso anterior. Realizamos un test de convergencia sobre W y H cada ciertas iteraciones, si hemos alcanzado unas matrices válidas para la precisión propuesta terminamos el proceso de actualización de matrices.

Para realizar el entrenamiento trabajamos con matrices del orden de 16384x900, debido a su tamaño y que son necesarias múltiples matrices auxiliares es necesario dividir la matriz original en varias porciones con las que realizaremos el cálculo. En consecuencia también es necesario realizar particiones de las matrices auxiliares que serán necesarias para el kernel de la tarjeta.

Debido a que nosotros necesitamos partir los datos, partimos V en 4, obtenemos un H y W aleatorias y en cada iteración dividimos H en cuatro, calculamos la acumulación por filas de W y la guardamos en X1. Ahora para cada uno de los cuartos realizamos una serie de productos de matrices, y operaciones punto a punto para obtener los nuevos valores de W y H, con los que probamos la convergencia, y si aún no es satisfactoria iniciamos otra iteración. En las Figuras 4.7 y 4.8 podemos ver una traza de nuestro algoritmo.

En la Figura 4.9 podemos observar los espacios de memoria requeridos en la tarjeta, en el caso de partir la matriz original en cuatro porciones y en el Cuadro 4.4 se muestran los tamaños de cada matriz de la memoria para un caso particular de 16000x12000.

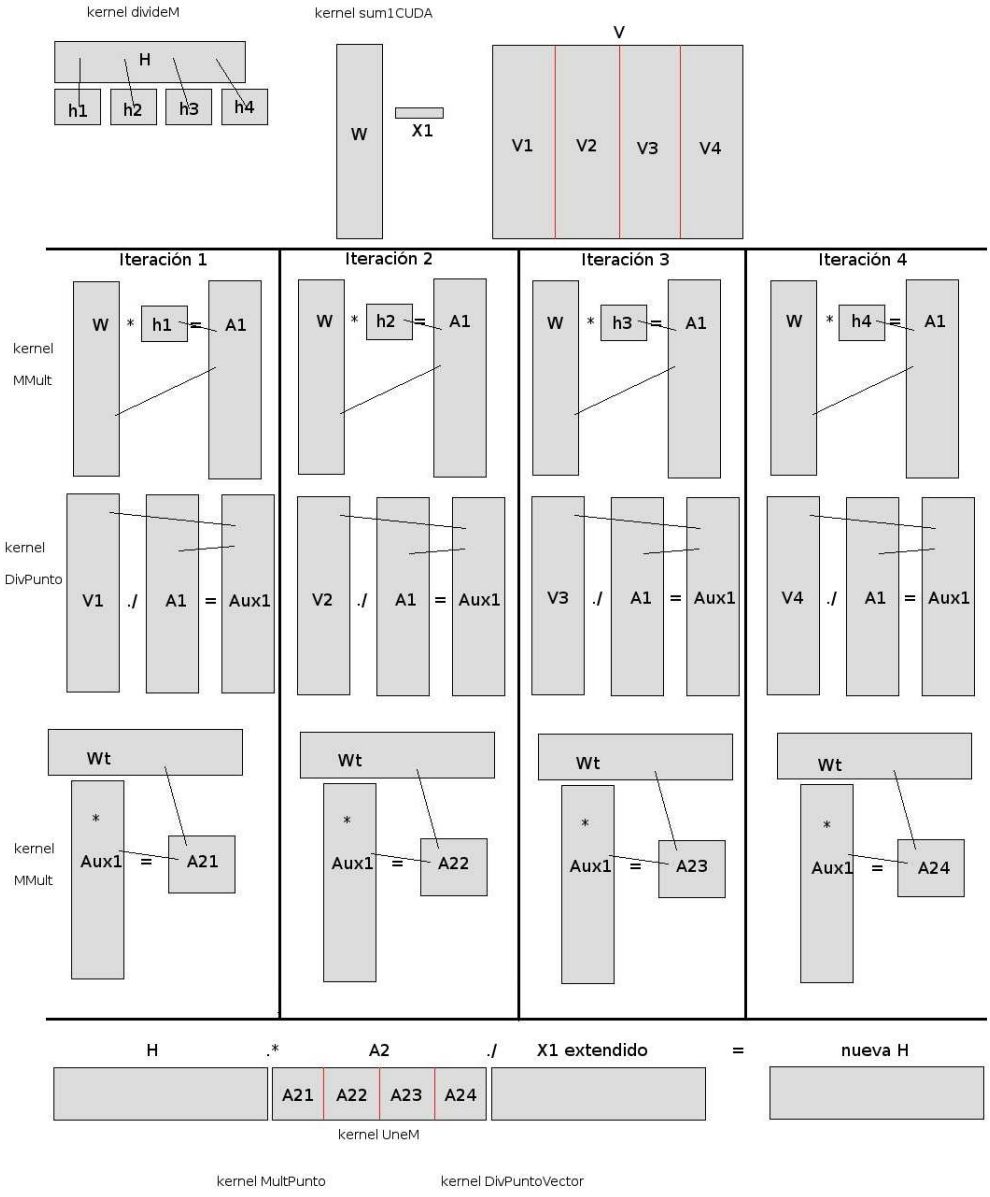


FIGURA 4.7: Traza del programa 1 de 2

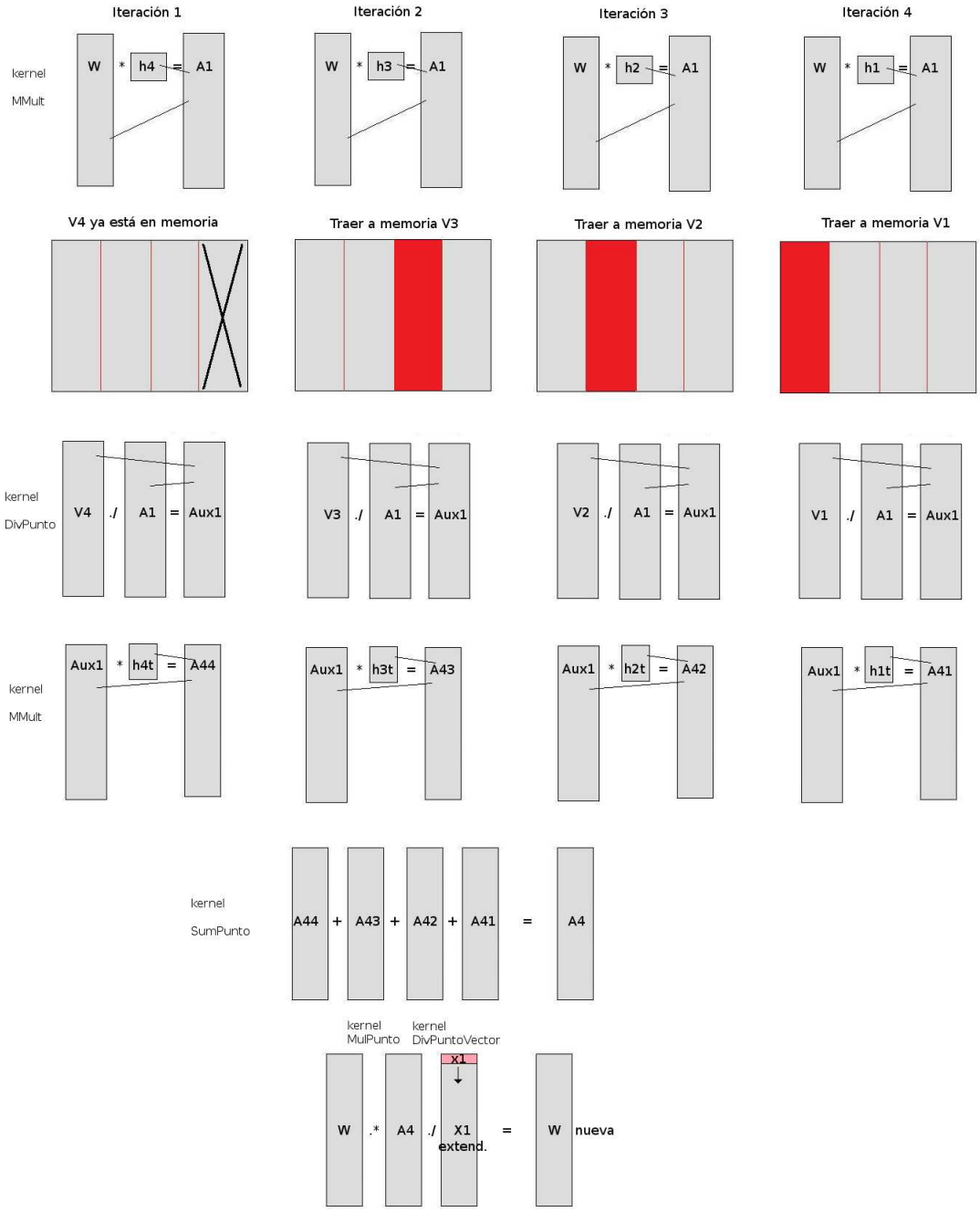


FIGURA 4.8: Traza del programa 2 de 2

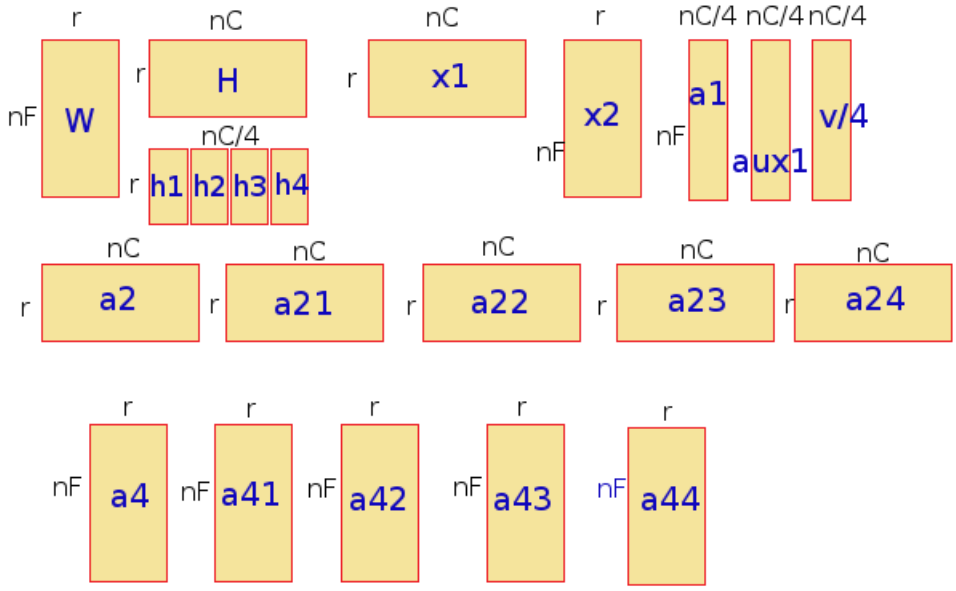


FIGURA 4.9: Mapa de Memoria de la Tarjeta, r mucho menor que nF,nC

Matriz	Tamaño	Espacio
V	nF x nC/4	183 MBytes
W	nF x r	12 MBytes
H	r x nC	9.2 MBytes
h1	r x nC/4	2.3 MBytes
h2	r x nC/4	2.3 MBytes
h3	r x nC/4	2.3 MBytes
h4	r x nC/4	2.3 MBytes
X1	r	0.002 MBytes
X2	r	0.002 MBytes
a1	nF x nC/4	183 MBytes
aux1	nF x nC/4	183 MBytes
a4	nF x r	12 MBytes
a41	nF x r	12 MBytes
a42	nF x r	12 MBytes
a43	nF x r	12 MBytes
a44	nF x r	12 MBytes
a2	r x nC	9.2 MBytes
a21	r x nC/4	2.3 MBytes
a22	r x nC/4	2.3 MBytes
a23	r x nC/4	2.3 MBytes
a24	r x nC/4	2.3 MBytes
Espacio total de 657 MBytes		

TABLA 4.4: Memoria requerida por las matrices con r = 200, nF = 16000 y nC = 12000

4.1.3.1. Optimizaciones

Debido a que las transferencias hacia y desde el dispositivo son muy costosas nos interesa minimizar el número de ellas, en nuestro caso, con cuatro trozos de la matriz original, y debido al funcionamiento del algoritmo, necesitaríamos realizar ocho transferencias, dos por cada cuarto de la matriz, en el siguiente orden: transferir V1, V2, V3, V4 y realizar una serie de operaciones con ellos, y volver a operar y por lo tanto transferir V1, V2, V3, V4.

Podemos ahorrar dos de esas transferencia invirtiendo el orden de cálculo de la segunda parte del algoritmo, de tal modo que transfiramos los cuartos del 1 al 4, y en la segunda parte lo hagamos del 4 al 1, aprovechando que el cuarto trozo ya está en memoria no es necesaria esa transferencia, y al final de la segunda parte queda en memoria el primer cuarto de la matriz, que puede ser aprovechado para la siguiente iteración.

Para hacernos una idea del coste de una transferencia basta decir que las transferencias son del orden de unos segundos, y en total todo el algoritmo tarda unos 12 segundos, por lo que evitar una transferencia reduce significativamente el tiempo empleado.

4.1.3.2. Convergencia

La convergencia de W y H de nuestro algoritmo NMF se evalúa cada diez iteraciones. La medida de convergencia es útil para ahorrar iteraciones si ya sabemos que W y H han llegado a una buena aproximación.

En esta sección de convergencia, lo primero que se hace es mirar si W o H en alguna iteración entre todas las operaciones han llegado a tener ceros en algún elemento de las matrices. Esto se lleva a cabo mediante un kernel en el que cada hilo comprueba si el elemento correspondiente al hilo es cero, y en tal caso lo sustituye por una cifra mínima. Una vez hecha esta comprobación, la medida de la convergencia se centra en la matriz H. Al depender la convergencia de W de la convergencia de H solo se realiza la comprobación de una de ellas para ahorrar tiempo de cálculo.

Una vez libre de ceros la matriz H, se obtiene de ella un vector con los números de fila en los cuales se encuentra el elemento máximo de cada columna. La base de esta comprobación de convergencia se basa en que este vector se mantenga muy similar a lo largo de varias comprobaciones. En el algoritmo original del que tomamos la idea se creaba una matriz de dispersión de ancho y alto del tamaño de este vector, con unos en las posiciones en las que la intersección del vector de la comprobación de convergencia actual, y el vector de la comprobación anterior contienen el mismo índice.

Si hubiéramos tratado esto como una matriz normal en vez de como una matriz de dispersión, habríamos necesitado reservar en la tarjeta el tamaño de toda esta matriz cuyo tamaño sería número de columnas de V al cuadrado, y teniendo un número fijo de columnas V por contener imágenes de tamaño 128x128, el tamaño de esta matriz ascendería a 1 Gbyte, por lo que habríamos necesitado partir la matriz para poder operar con ella en la tarjeta.

En una primera aproximación teniendo en cuenta que se generaban matrices de dispersión, tomábamos los dos vectores que conformarían la matriz, y primero calculábamos el número de elementos no nulos que tendría ésta, ya que con CUDA no se puede reservar memoria desde un kernel dentro de la propia tarjeta.

Una vez conocido el número de elementos no nulos que conformarían la matriz, se procede a reservar memoria para un vector de pares de enteros, que representan las coordenadas de los elementos no nulos dentro de la matriz de dispersión. De esta forma ya conseguíamos una reducción del tamaño en la representación de esta matriz del 80 %.

Una vez obtenido el vector de pares de enteros, se compara con el vector del cálculo de la convergencia anterior con otro kernel. Aquí se presentó un problema ya que la representación

de una misma matriz de dispersión de esta forma podía generar múltiples posibilidades debido a que las coordenadas se iban introduciendo sin orden en el vector, así que procedimos a ordenar el vector primero por la primera coordenada, y después por la segunda. Además nos dimos cuenta de que la matriz de dispersión que se genera es simétrica, por lo que sólo necesitábamos generar las coordenadas de la parte superior a la diagonal principal de la matriz. Con esto el tamaño en la representación de la matriz disminuyó a la mitad.

Como el objetivo final de este proceso era obtener el número de elementos distintos entre las dos matrices de dispersión, finalmente optamos por otra alternativa. Cogiendo los vectores de elementos máximos de cada columna de la anterior iteración y la actual, creamos un kernel que directamente comparaba los dos vectores. Como la variable donde se iba incrementando el número de elementos distintos entre los dos vectores era única, y todos los hilos del kernel debían acceder a ella, el acceso a esta variable debe hacerse en modo exclusivo, limitando el paralelismo. En CUDA existen unas funciones atómicas que facilitan este trabajo, pero tiene ciertas limitaciones, por lo que implementamos un mutex para realizar este acceso exclusivo. Esto ralentiza la ejecución global del algoritmo, pero sigue compensando puesto que para una comparación en la que se tendría que entrar en la zona del mutex, había cientos de comparaciones en las que no se entraba, y esto sí se realiza de forma paralela. Implementación del mutex:

```
while(waiting)
                {
                    if(atomicExch(mutex,0))
                    {
                        atomicAdd(cont,1);
                        waiting = 0;
                        atomicExch(mutex, 1);
                    }
                }
```

Por lo tanto podemos decir que de la implementación inicial a la final existe una reducción de reserva de memoria de prácticamente el 100 %, ya que sólo almacenamos el vector de elementos máximos de cada columna, que ocupa unos 64 Kbytes. Cuando estos dos vectores no cambian de una comprobación a otra, es decir, cuando el resultado de la comparación es cero, procedemos a ir incrementando un contador que se tendrá en cuenta a la hora de la terminación del algoritmo. Si este contador llega al número de comprobaciones que hemos establecido como índice de que se ha alcanzado la convergencia, y que se pasa a la función por parámetro, el algoritmo para y genera el resultado, aunque no se haya alcanzado el número máximo de iteraciones especificado.

4.1.3.3. Proyección de caras

Una vez terminado el algoritmo NMF, con W y H finales generados, necesitamos transformar estos datos para que sean útiles para generar un proyector. En nuestro caso nos quedamos con W, puesto que forma una base de las características identificativas de las caras que se han entrenado con V, que casi siempre va a ser una matriz rectangular. Para identificar una cara es necesario proyectarla sobre la base que hemos tomado, para lo que multiplicamos el vector cara por una matriz de proyección.

Debemos transformar W para obtener un proyector con el que operar. Para ello necesitamos calcular la inversa de esta matriz, pero al ser una matriz rectangular no podemos usar los métodos tradicionales.

En nuestro caso utilizamos el método de la pseudo-inversa de Moore-Penrose [Moo20][Pen55]. Este método es una generalización del cálculo de la inversa de matrices. Para esto, hicimos uso de un paquete de funciones de álgebra lineal llamado LAPACK[1] [lap], escrito en For-

[1] LAPACK son las siglas de Linear Algebra PACKage

tran90, con una función específica del cálculo de la pseudo-inversa, la función pinv. Esta función no puede ser llamada directamente desde C, por lo que hicimos una implementación de pinv partiendo de otra función de este paquete llamado dgelss.

Para utilizar las funciones de Fortran necesitamos realizar un enlazado desde C, y transformar los parámetros para poder introducirlos en la función. El resultado de esta función es una matriz que se guarda en memoria para posteriormente utilizarla como proyector, multiplicando esta pseudo-inversa por la imagen transformada en un vector columna, obteniendo así una transformación de la imagen que podrá ser comparada con otras más fácilmente calculando distancias generadas por el proyector entre ellas.

4.1.4. Resultados

Trabajamos con una Nvidia GeForce 280GTX [NVIa], dispone de 1GB de memoria. La gama 280 dispone de más procesadores escalares que las versiones previas, proporcionando hasta 240 procesadores multihilo, lo que nos permite aumentar el grado de paralelismo. Podemos ver las especificaciones técnicas en la Figura 4.10

GPU Engine Specs:

Processor Cores	240
Graphics Clock (MHz)	602 MHz
Processor Clock (MHz)	1296 MHz
Texture Fill Rate (billion/sec)	48.2

Memory Specs:

Memory Clock (MHz)	1107 MHz
Standard Memory Config	1 GB
Memory Interface Width	512-bit
Memory Bandwidth (GB/sec)	141.7

FIGURA 4.10: Especificaciones técnicas de la tarjeta GeForce 280 GTX

Soporta hasta a versión CUDA 1.3, es decir, puede utilizar todas las funcionalidades del lenguaje, entre las cuales resultan muy interesantes las operaciones atómicas, no soportadas en versiones inferiores.

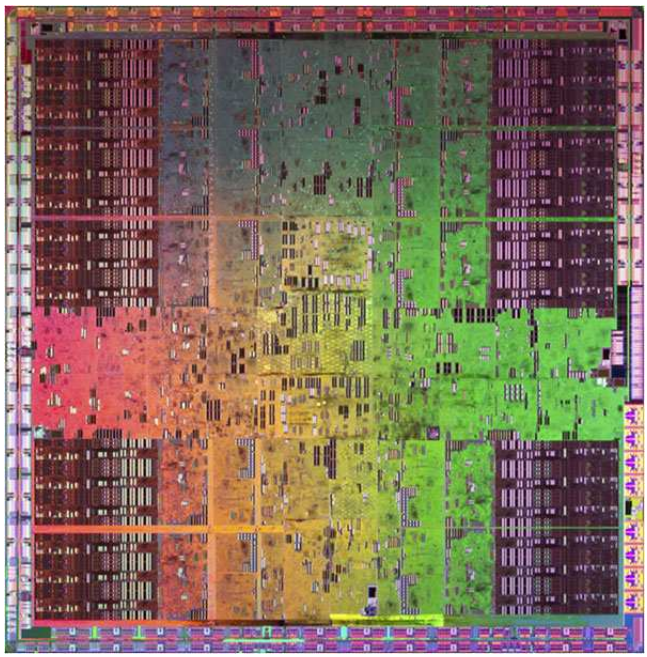
4.1.4.1. Rendimiento

Creamos una versión del algoritmo en C, que fue la que utilizamos en un principio para guiarnos para la primera implementación en CUDA. A partir de esa implementación en C, hicimos una serie de optimizaciones con herramientas específicas para algunos tipos de operaciones.

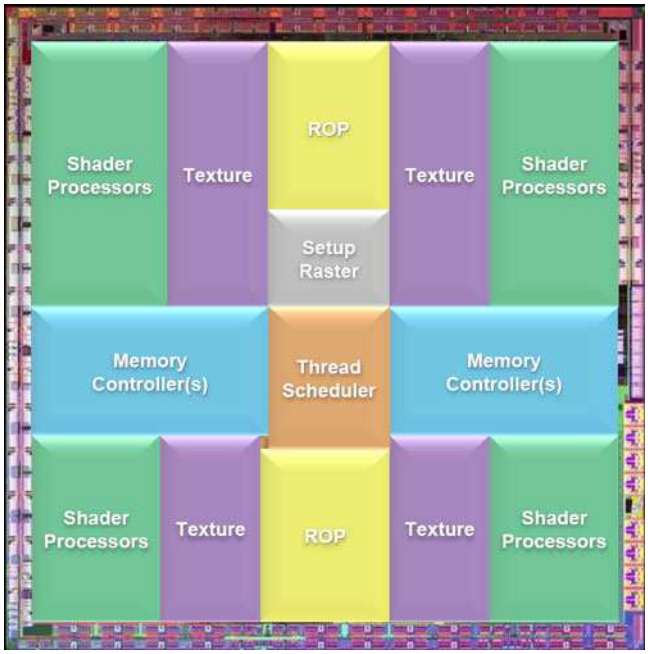
Para las multiplicaciones de matrices utilizamos la librería ATLAS[2] [atl], que nos proporciona de interfaces para una implementación eficiente de BLAS[3] [bla], una librería que provee

[2] ATLAS son las siglas de Automatically Tuned Linear Algebra Software.

[3] BLAS son las siglas de Basic Linear Algebra Subprograms



(a) Nvidia GeForce 280 GTX



(b) Distribución de las unidades

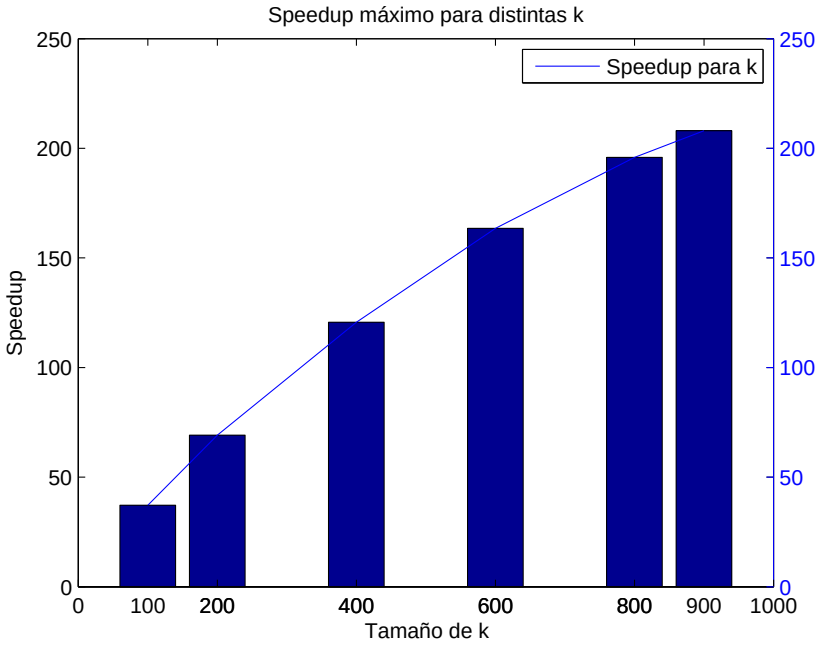


FIGURA 4.11: Speedup CPU/GPU obtenido en relación a k.

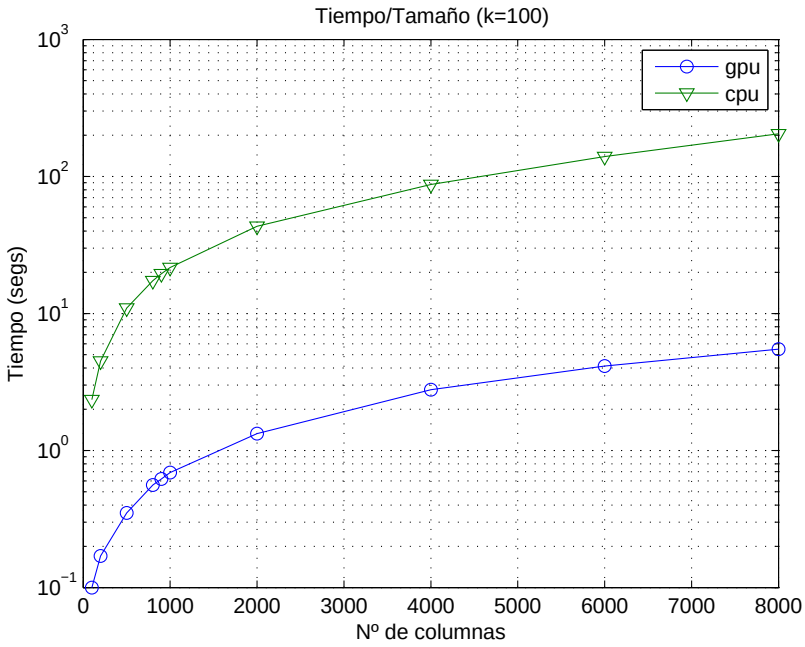
de operaciones de álgebra lineal, como multiplicación de vectores y matrices, así como de algunas rutinas de LAPACK, una librería de álgebra lineal mucho más completa.

Asimismo intentamos explotar al máximo el paralelismo en bucles con hilos, para lo que utilizamos OpenMP[4] [Ope] y así ocupar todos los núcleos posibles con trabajo útil.

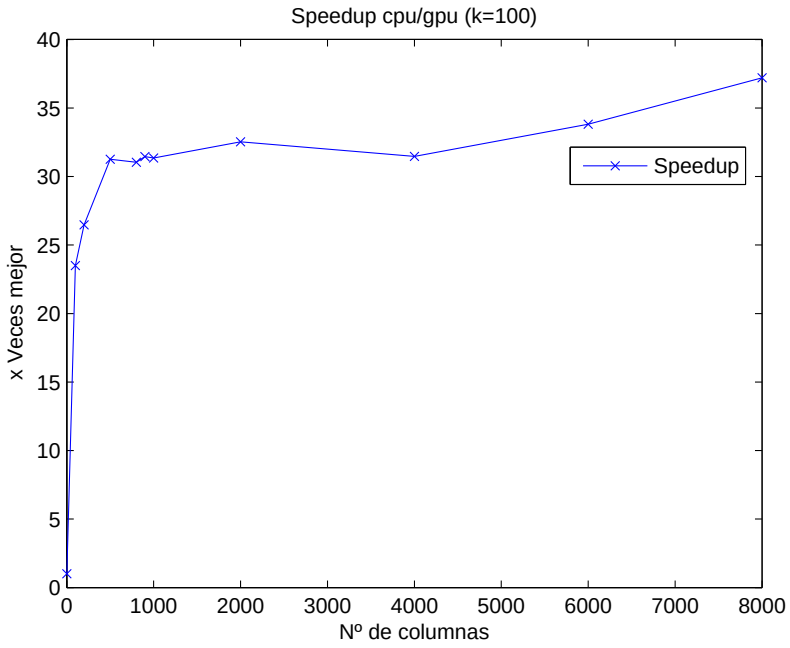
Para realizar la comparación en tiempo de los dos algoritmos hemos tomado medidas del tiempo de ejecución de una iteración. Los tiempos de ejecución de una iteración para cada uno de los dos algoritmos son muy estables, pero aún así hemos tomado los datos de los tiempos de 20 iteraciones y hemos hecho una media, para tener un tiempo significativo. Hemos fijado para estas medidas el número de filas de la matriz V a 16384, que es el tamaño estándar de las imágenes normalizadas, y por lo tanto es la longitud del vector columna de V. Para el número de filas hemos realizado ejecuciones desde un tamaño de 100 hasta un tamaño de 8000. Los datos son más significativos para un número mayor de columnas, ya que el entrenamiento con este algoritmo es más eficaz con mayor número de imágenes. Para estos tamaños de filas hemos realizado ejecuciones fijando una k en varios valores desde 100 hasta 900, llegando a la ocupación máxima de la tarjeta en memoria.

4.1.4.1.1. Exploración del valor de k Hemos realizado pruebas para ver el efecto del tamaño de k en la eficiencia del algoritmo. Hemos tomado valores de k desde 100 hasta 900, comprobando los tiempos de ejecución utilizando matrices V con 16000 filas, y un número variable de columnas.

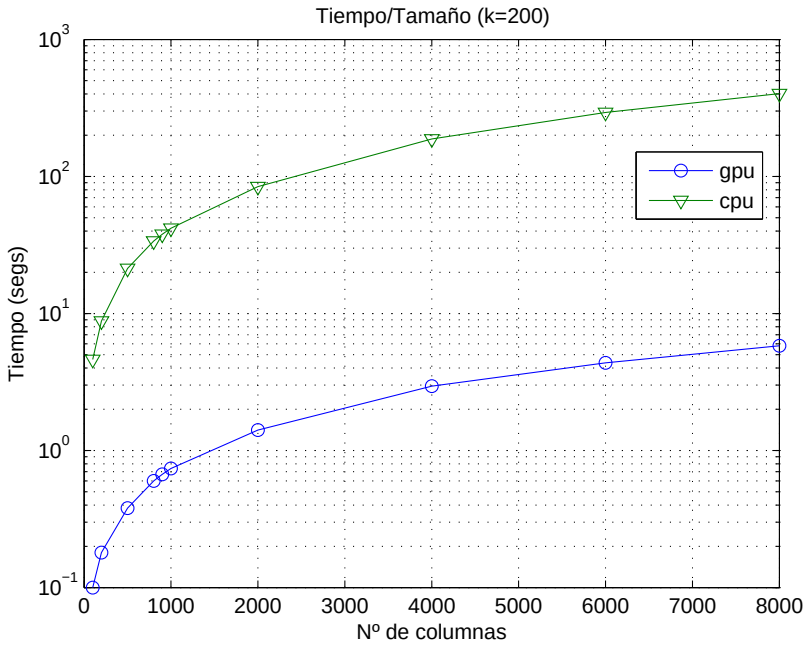
[4] OpenMP es una API que permite añadir concurrencia a las aplicaciones mediante paralelismo con memoria compartida. Se basa en la creación de hilos de ejecución paralelos compartiendo las variables del proceso padre que los crea.



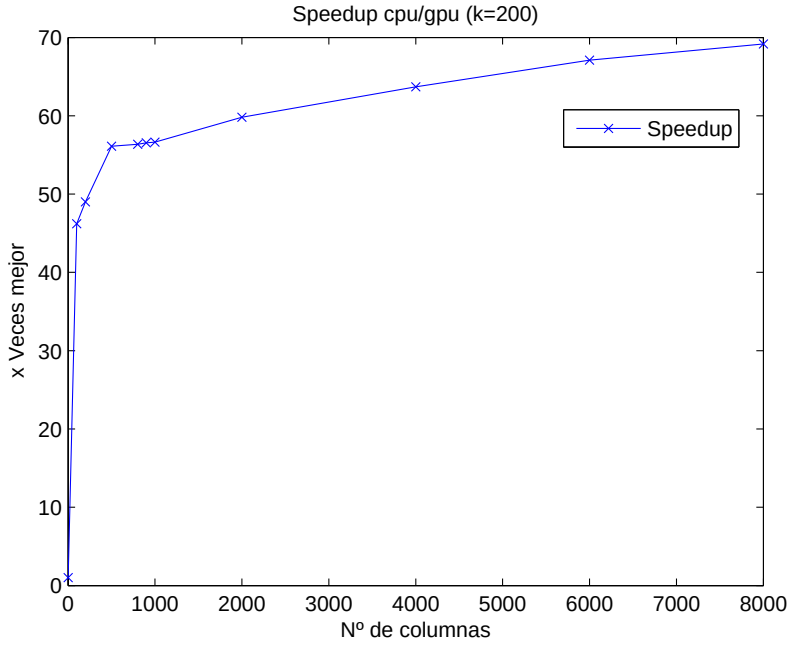
(a) Comparación de tiempos del algoritmo en CPU y GPU para tamaño de k=100.



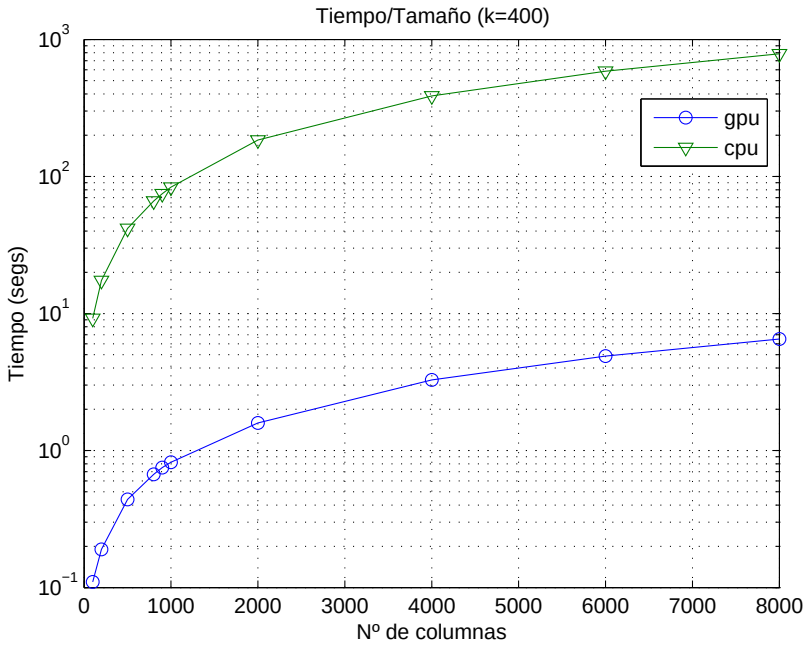
(b) Speedup del algoritmo en GPU frente al de CPU para tamaño de k=100.



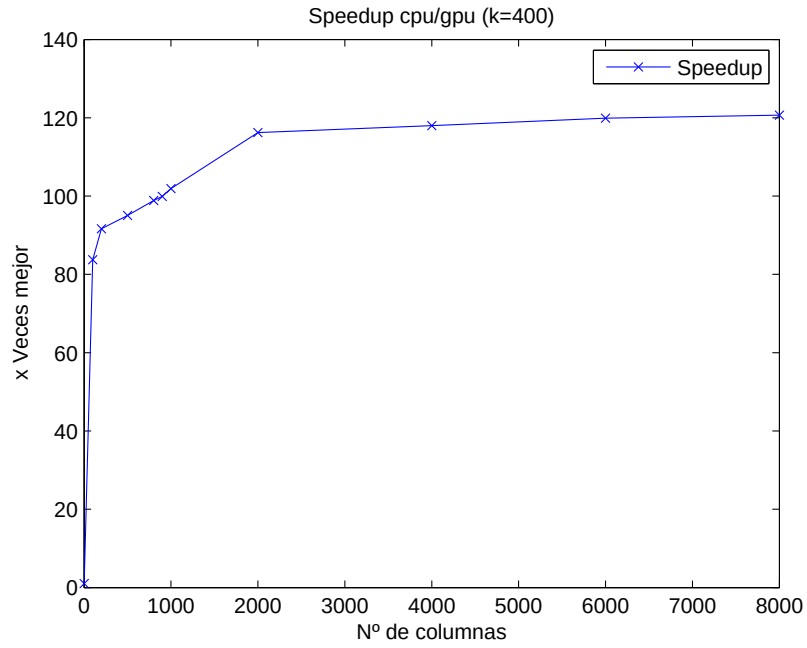
(c) Comparación de tiempos del algoritmo en CPU y GPU para tamaño de k=200.



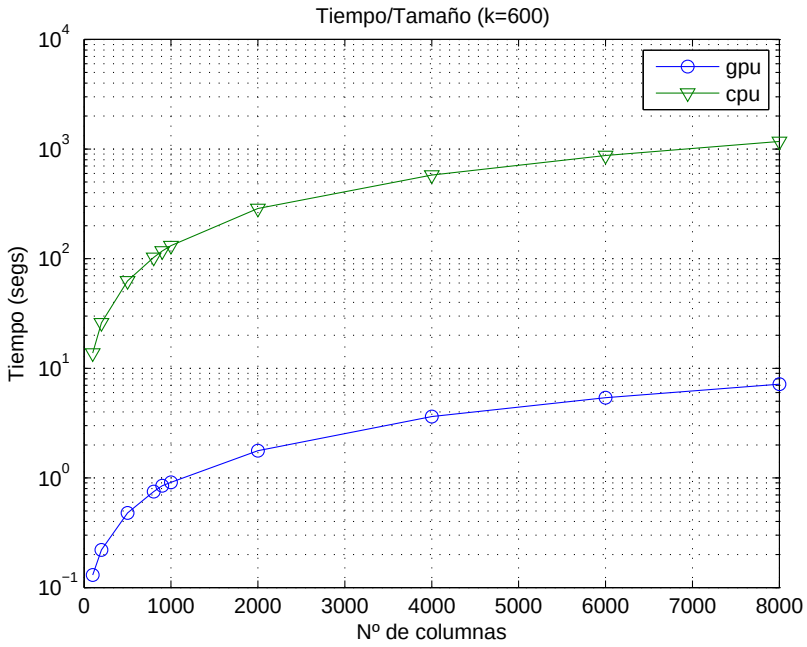
(d) Speedup del algoritmo en GPU frente al de CPU para tamaño de k=200.



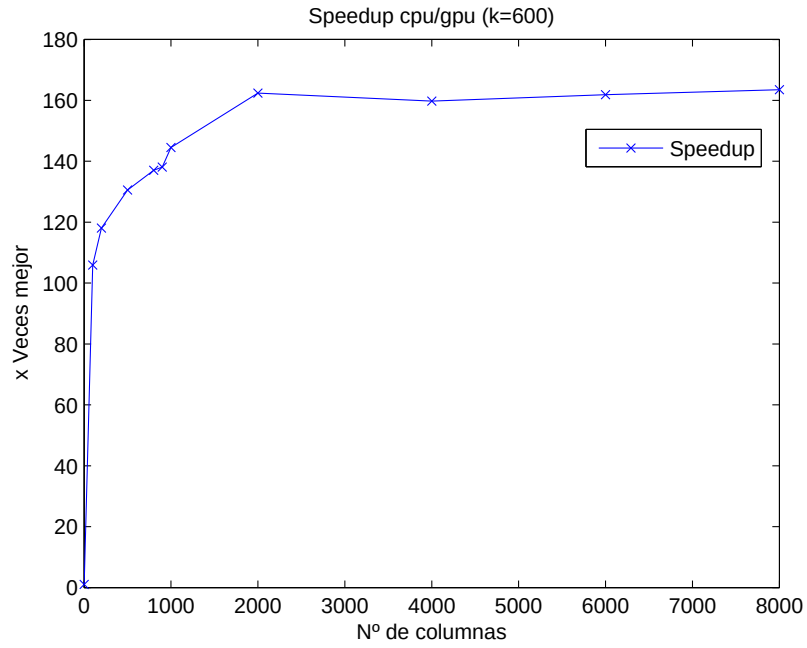
(e) Comparación de tiempos del algoritmo en CPU y GPU para tamaño de k=400.



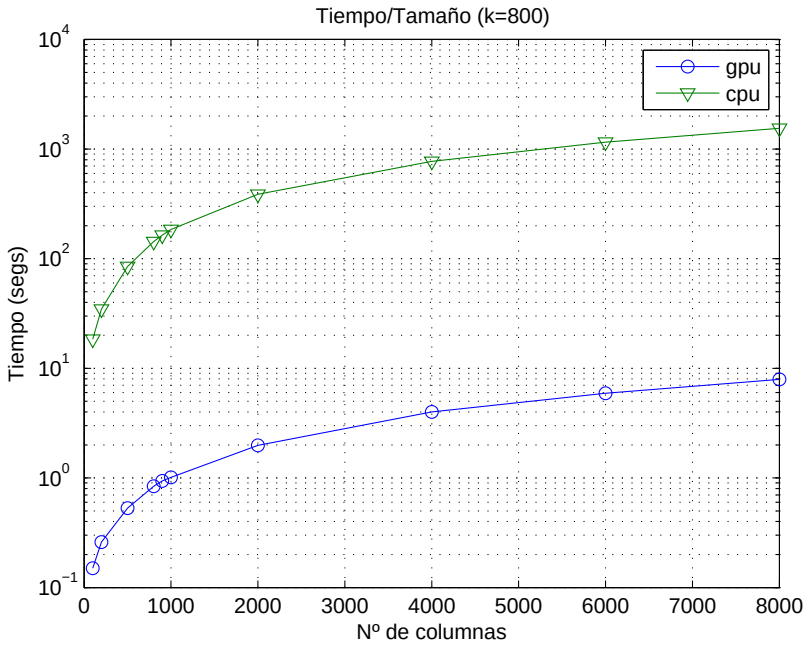
(f) Speedup del algoritmo en GPU frente al de CPU para tamaño de k=400.



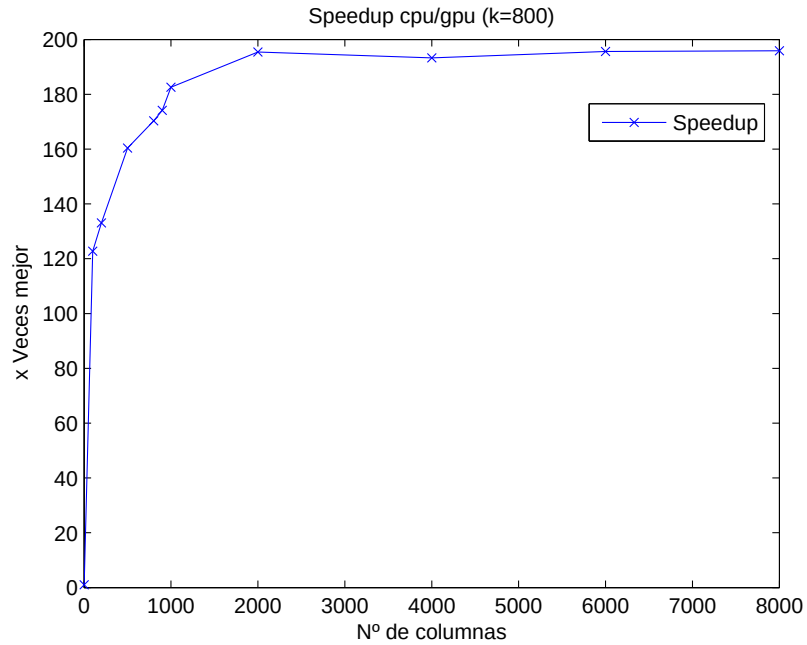
(g) Comparación de tiempos del algoritmo en CPU y GPU para tamaño de k=600.



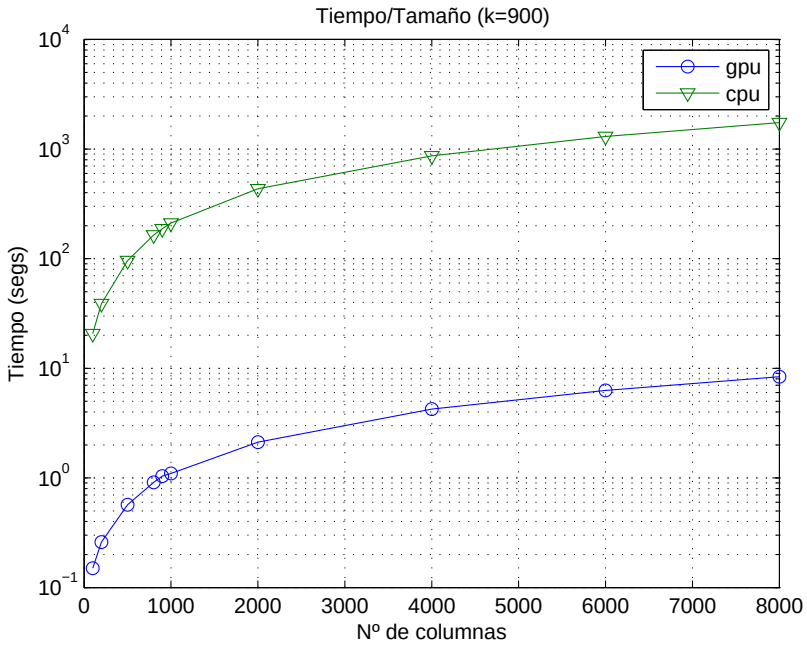
(h) Speedup del algoritmo en GPU frente al de CPU para tamaño de k=600.



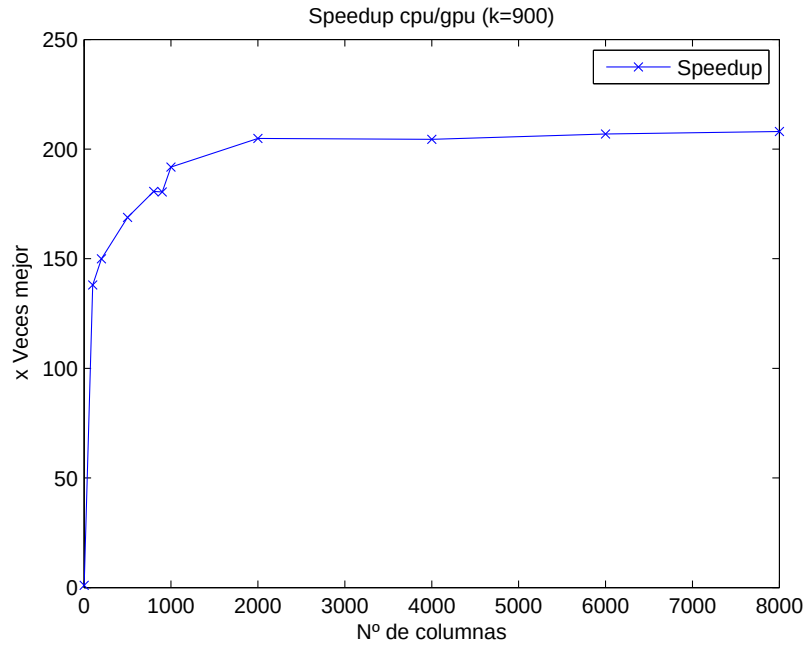
(i) Comparación de tiempos del algoritmo en CPU y GPU para tamaño de k=800.



(j) Speedup del algoritmo en GPU frente al de CPU para tamaño de k=800.



(k) Comparación de tiempos del algoritmo en CPU y GPU para tamaño de k=900.



(l) Speedup del algoritmo en GPU frente al de CPU para tamaño de k=900.

Como podemos observar, el algoritmo implementado para CPU escala linealmente, mientras que el implementado para GPU apenas varía. Por lo tanto, la diferencia entre las dos implementaciones, a mayor k o número de columnas, es mucho más acusada, lo que se traduce en un mayor speedup, como podemos observar en la Figura 4.11.

Las pruebas realizadas con el tamaño mayor de k son en las que más diferencia hemos podido observar. El speedup máximo obtenido ha sido de x208. En la Subfigura 4.12(l) podemos observar que aumentando el número de columnas se produce un incremento del speedup muy importante hasta número de columnas 1000. A partir de ese momento el speedup se mantiene casi constante, llegando siempre al máximo cuando la ocupación de memoria en la tarjeta gráfica es mayor.

Otro de los factores determinantes a la hora de conseguir un mayor speedup es el tipo de tarjeta utilizada. En un principio utilizamos una geForce 8800gtx con 768Mbytes de RAM, y en las pruebas realizadas con los k y número de columnas mayores se desbordaba la memoria, obteniendo unos resultados algo peores, llegando a un speedup de x103, siendo la subida de éste más gradual, creciendo de forma abrupta hasta número de columnas 2000. Con la geForce 280gtx, con 1Gbyte de RAM, la memoria de la tarjeta no se llegó a desbordar en ningún momento, ya que el algoritmo estaba preparado en un principio para tarjetas con 1Gbyte de memoria RAM.

4.1.4.2. Eficacia del algoritmo para reconocimiento facial

Al igual que en los resultados de tiempo para el algoritmo, en los resultados para reconocimiento facial debemos introducir primero el tipo de medición que hemos llevado a cabo, para poder comparar estos resultados con otros similares.

Nuestros experimentos se basan en los llevados a cabo para la FRGC[5] [FRG], una competición que tiene como finalidad promover los avances en tecnología de reconocimiento facial. Consta de una serie de elementos puestos a disposición de las distintas entidades participantes: una base de datos multimodal llamada FRGC, un protocolo de evaluación y una herramienta software para la comparación de resultados. Consiste en seis experimentos diseñados para medir la calidad de los sistemas de verificación facial en condiciones de iluminación controladas y no controladas, disponiendo de información tridimensional, de una o varias imágenes, y de ambos tipos de datos: tridimensionales y color.

La base de datos FRGC consta de información tridimensional y de imágenes de alta resolución, tomadas en diferentes condiciones de iluminación, y con varias expresiones faciales. Está formada por 200 individuos, con diferentes sesiones de capturas realizadas a lo largo de once semanas, durante dos años académicos. Cada sesión de captura está compuesta por cuatro imágenes tomadas en condiciones controladas, dos adquiridas en condiciones no controladas, y una captura tridimensional. En la Figura 4.12 puede verse una sesión completa de un individuo. Las imágenes adquiridas en condiciones controladas fueron tomadas bajo dos iluminaciones diferentes (con dos o tres luces de estudio) y con dos expresiones faciales distintas (neutral y sonrisa). Las imágenes tomadas bajo condiciones no controladas fueron adquiridas bajo una iluminación variable (en un escenario exterior, en un pasillo y en un patio) y con los mismos gestos (neutral y sonrisa).

El conjunto de datos está dividido en dos particiones diferentes: conjunto de entrenamiento y conjunto de validación. El primero de ellos consta de 12776 imágenes de 222 sujetos diferentes, y de 943 capturas tridimensionales. El segundo conjunto está formado por imágenes de 446 sujetos adquiridas en 4007 sesiones diferentes.

[5]FRGC son las siglas de Face Recognition Grand Challenge

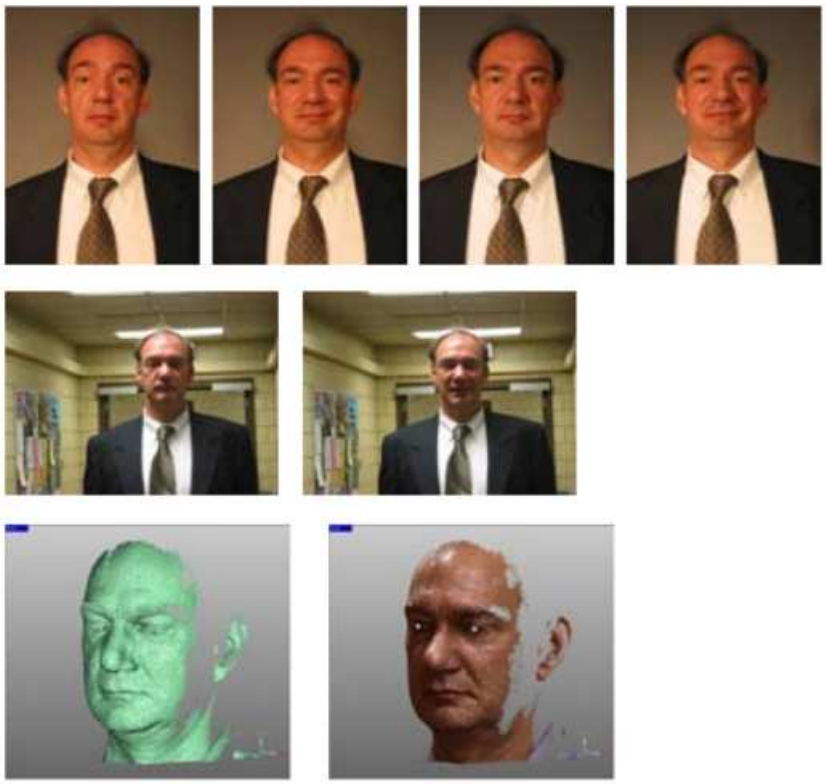


FIGURA 4.12: Ejemplo de sesión completa de un individuo.

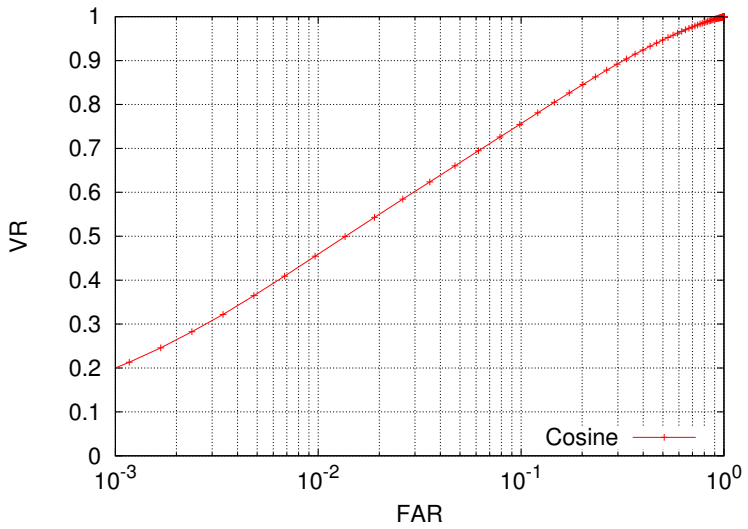


FIGURA 4.13: Curva ROC para el experimento 1 de FRGC con NMF (k=300).

La resolución de las imágenes es de 1704 x 2272 píxeles, o de 1200 x 1600 píxeles, adquiridas con cámaras de alta resolución.

De los experimentos propuestos hemos llevado a cabo dos. En el experimento 1, las imágenes para entrenamiento son imágenes en condiciones controladas (una por persona), y las imágenes de validación también son imágenes en condiciones controladas, es decir, este es el experimento control. El otro experimento que llevamos a cabo es el 4. En este experimento, las imágenes para entrenamiento son imágenes bajo condiciones controladas, y el set de prueba consiste en imágenes bajo condiciones no controladas.

Los resultados de estos experimentos para diferentes algoritmos se miden mediante una curva ROC[6], curva en cuyo eje de abscisas se representa la sensibilidad frente a falsos positivos, lo que técnicamente se denomina FAR[7], y en cuyo eje de ordenadas se representa la tasa de aciertos (VR[8] en las gráficas).

Nosotros hemos utilizado el algoritmo NMF para entrenar una parte de las imágenes para entrenamiento, 900 en total. Hemos ejecutado el algoritmo tomando como parámetros k=300 y número de iteraciones 100000, lo que ha llevado unas 18 horas de ejecución. Hemos tomado la matriz W resultante, hemos calculado su pseudo-inversa, y la hemos guardado como proyector para posteriormente utilizarlo para realizar la prueba. Podemos observar en la Figura 4.13 como en el experimento de referencia la curva observada es bastante mala, necesitando aumentar el FAR demasiado para obtener un VR mediocre, por lo que la tasa de fallos es muy alta. En el experimento 4, cuyos resultados podemos observar en la Figura 4.14, vemos que la curva ROC es aun peor, concluyendo que por lo menos con este método de entrenamiento, y con el proyector obtenido, NMF no sirve directamente para reconocimiento facial.

Al ver estos resultados decidimos volver a entrenar otro proyector, esta vez con un k de 500. Ejecutamos también durante 100000 iteraciones, lo que llevó algo más de 24 horas. Podemos observar en la gráfica de la Figura 4.15 una mejora considerable, aunque no suficiente

[6]ROC son las siglas de Receiver Operating Characteristics.
[7]FAR son las siglas de False Acceptance Rate
[8] VR son las siglas de Verification Rate.

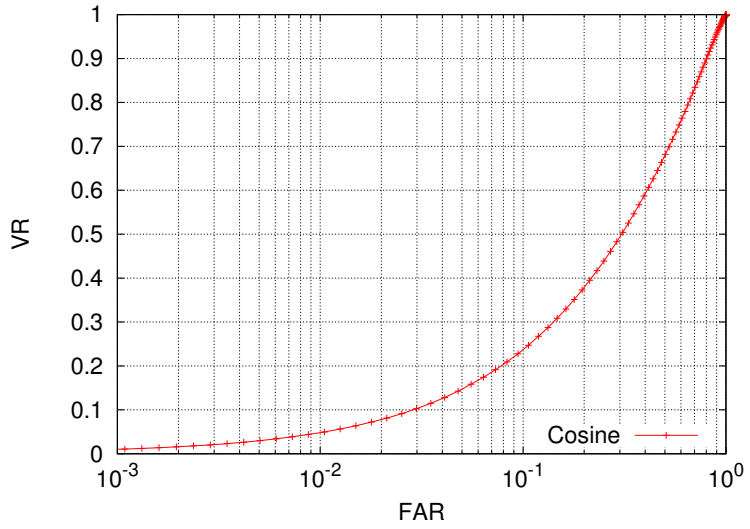


FIGURA 4.14: Curva ROC para el experimento 4 de FRGC con NMF (k=300).

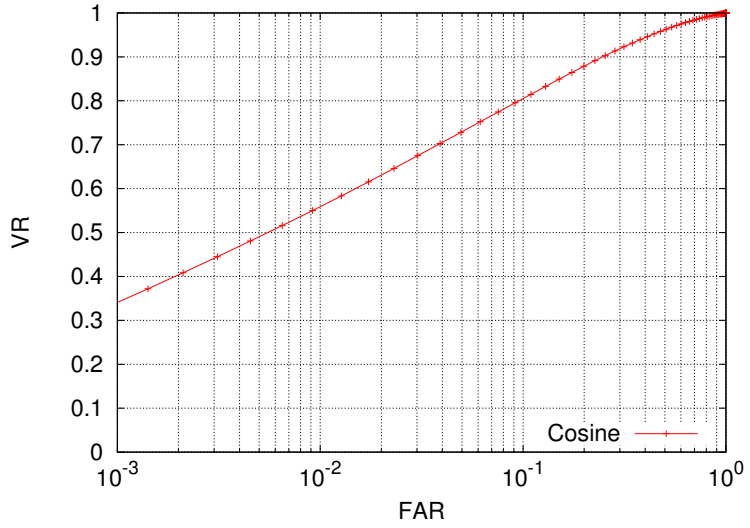


FIGURA 4.15: Curva ROC para el experimento 1 de FRGC con NMF (k=500).

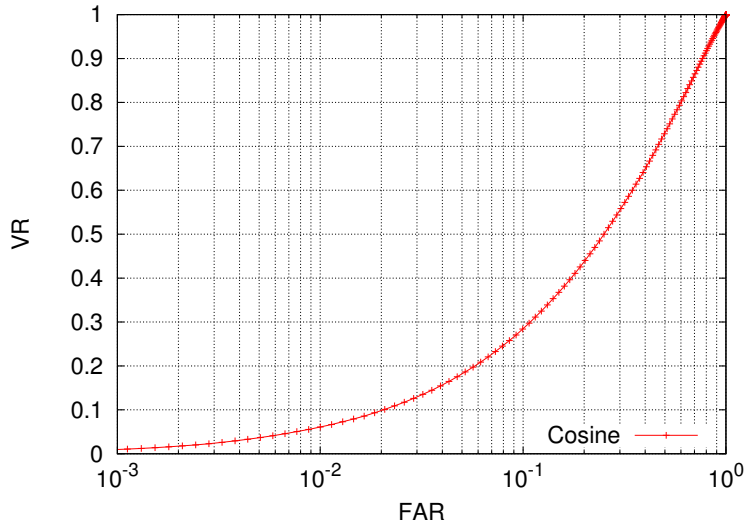


FIGURA 4.16: Curva ROC para el experimento 4 de FRGC con NMF (k=500).

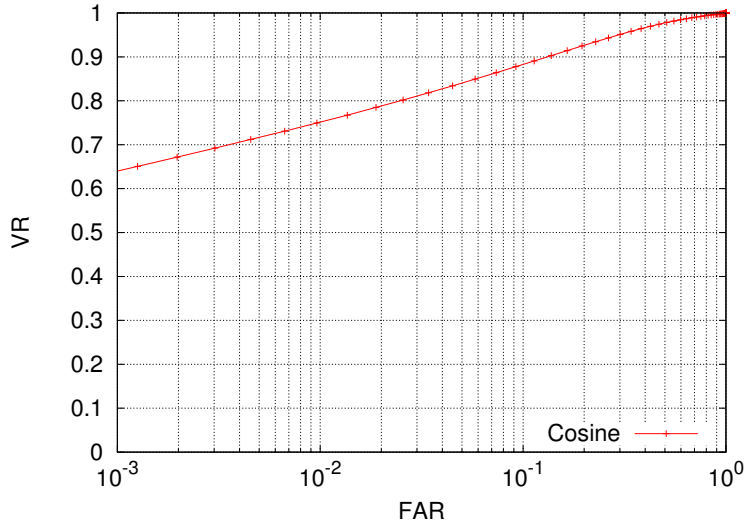


FIGURA 4.17: Curva ROC para el experimento 1 de FRGC con NMF + Gabor (k=500).

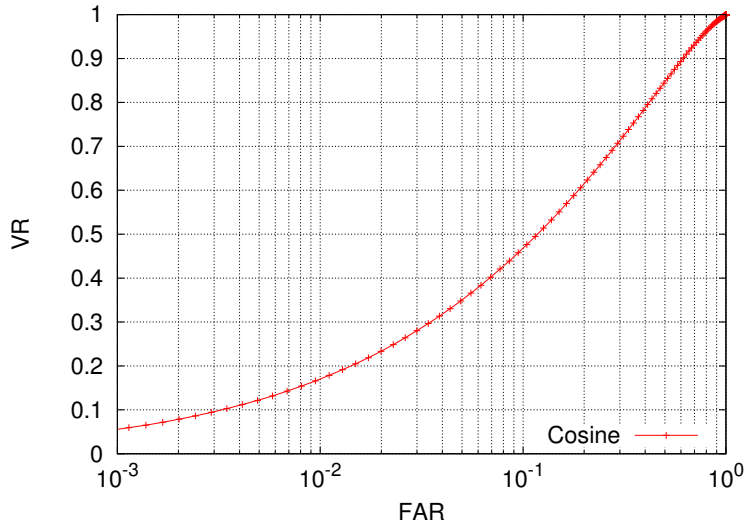


FIGURA 4.18: Curva ROC para el experimento 4 de FRGC con NMF + Gabor (k=500).

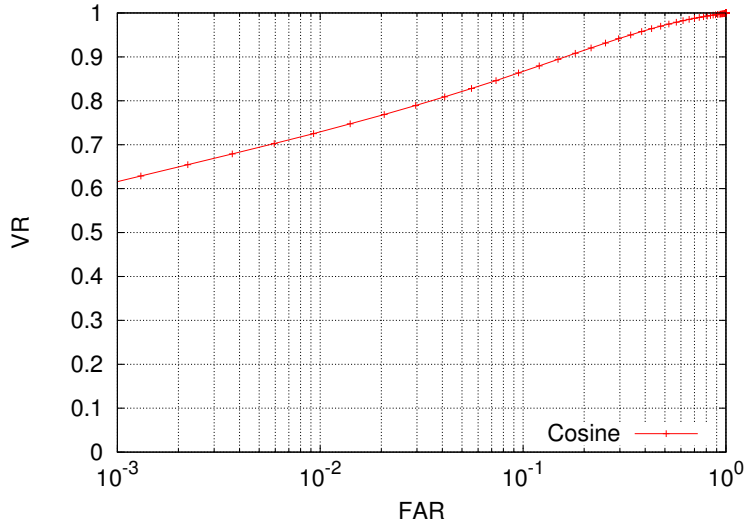


FIGURA 4.19: Curva ROC para el experimento 1 de FRGC con NMF + Gabor (k=1000).

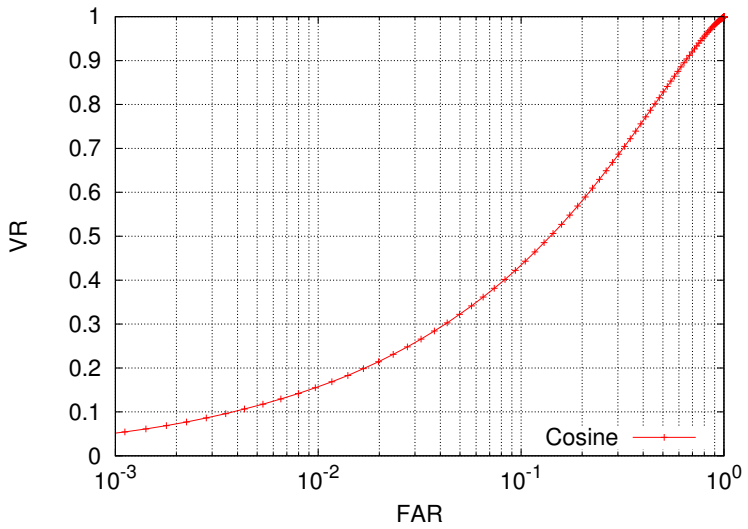


FIGURA 4.20: Curva ROC para el experimento 4 de FRGC con NMF + Gabor (k=1000).

como para poder considerarse apta para el reconocimiento facial. En lugar de empezar con una tasa de aciertos del 20 % para un FAR muy pequeño, en esta ocasión logramos una tasa de aciertos de más o menos el 33 %, por lo que podemos establecer una relación entre la k utilizada y la bondad del proyector a la hora de reconocer caras. Obviamente los únicos parámetros que podemos modificar son esta k y el número de iteraciones para la ejecución del algoritmo. Un número mayor de iteraciones nos lleva también a una mejor aproximación, por lo tanto también podemos deducir que entrenando con la misma k durante más tiempo, considerando los valores de W y H iniciales iguales al inicio, nos llevará a producir un proyector también algo mejor.

De todas formas podemos observar que en el experimento 4, con su curva ROC visible en la Figura 4.16, los resultados, aunque algo mejores, siguen siendo bastante malos como para poder utilizarse.

En un último intento, decidimos realizar una transformada de Gabor para las imágenes normalizadas, entrenando primero un proyector con las imágenes en columnas ya transformadas, y luego transformando las imágenes a comparar para poder multiplicarlas por el proyector. Primero generamos un proyector con 900 imágenes transformadas, utilizando una k de 500 y ejecutando durante 100000 iteraciones, para poder comparar resultados. Podemos decir que el filtrado de las imágenes ha mejorado considerablemente el uso del algoritmo NMF para reconocimiento facial. En el experimento 1, como podemos observar en la Figura 4.17, los resultados han mejorado bastante, iniciando la curva ROC en un valor casi el doble que con los otros proyectores. Aún así, en el experimento 4, aunque se ha notado una leve mejora, como se puede observar en la Figura 4.18, no es suficiente como para poder decir que este método pueda servir para reconocimiento facial. Generamos un último proyector con 8000 imágenes transformadas, utilizando una k de 1000 y ejecutando durante 50000 iteraciones. Este proyector tardó en generarse 3 días, ya que una iteración con tales tamaños de matrices tarda unos 5.7 segundos. Los resultados no han sido los esperados, ya que pensamos que con un k mayor y utilizando más imágenes, las curvas serían algo mejores que con el otro proyector utilizando Gabor, pero se puede observar en las Figuras 4.19 y 4.20 que las curvas son un poco peores. Esto se debe a que el proyector

no se pudo entrenar el tiempo suficiente para llegar a una aproximación buena de W, ya que añadiendo más imágenes se crea una V mucho mayor, y el tiempo de entrenamiento depende en gran medida del tamaño de V.

Estos resultados contradicen los obtenidos en el artículo [Gui02], basados en este otro artículo [LS99] de D. Lee y H. Seung, donde se muestra el método NMF como una técnica a tener en cuenta para el problema de reconocimiento de caras capturadas bajo condiciones no favorables, como cambios en expresiones o en la iluminación, obteniendo unos resultados de VR del %70, aunque no se llega a decir con qué FAR se obtienen estos resultados.

En dicho artículo también se menciona que se hizo una preclasificación de las caras según el género para obtener mejores resultados. Esta claro que las imágenes utilizadas en el entrenamiento hacen que se generen mejores o peores resultados, y en nuestro caso utilizamos imágenes de personas de ambos sexos, y de diferentes colores de piel. Para comparar nuestros resultados utilizamos la base de datos de FRGC, que ha sido ampliamente utilizada en la investigación de los métodos de reconocimiento facial, y teniendo en cuenta los resultados finales, podemos decir que el método NMF no es eficiente para el uso en reconocimiento facial.

4.2. Método NJIT

Sobre el código en C que disponemos de NJIT hemos paralelizado los procedimientos que tratan la proyección, creando para ello unos kernels sobre la tarjeta gráfica que se encarguen de realizar aquellas operaciones aptas para el modelo de programación de CUDA.

En concreto trabajamos con las funciones CenterTestGramMatrix y CenterTestGramMatrixKncRed de NJITUtils, con la función sobrecargada GramMatrix de Kernel, y con la función KernelFunc de GaussKernel.

Estas funciones trabajan sobre matrices, y las operaciones que realizan no tienen dependencia de datos entre ellas, así que son ideales para realizarlas en la tarjeta gráfica y obtener un buen rendimiento. Nuestra labor ha consistido en repartir los datos de modo que los accesos sean lo más eficientes posibles y hacer que los hilos realicen el cálculo intensivo sobre estos datos.

En este apartado haremos una descripción detallada de los kernel creados para el algoritmo NJIT.

4.2.1. CenterTestGramMatrix

La función CenterTestGramMatrix y su variante CenterTestGramMatrixKncRed tienen la tarea de generar una matriz Bc a partir de dos matrices Bnc y Knc. En el caso de la función CenterTestGramMatrixKncRed, la matriz Knc viene ya reducida y en cada posición almacena la media de los elementos de la columna correspondiente. En el caso de la función CenterTestGramMatrix deberemos hacer la reducción dentro de la función.

El primer paso es realizar la reducción de la matriz Knc para lo cual creamos un kernel que realiza la reducción de la matriz en paralelo, denominado rowRedCUDA.

A partir de aquí ambas funciones realizan las mismas operaciones. Primero realizamos la misma reducción con la matriz Bnc y almacenamos el vector resultado en BncRowRed. Posteriormente se calcula la media de todos los elementos de KncRowRed que guardamos en KncRed.

El siguiente paso es generar la matriz Bc a partir de las dos matrices y las reducciones que hemos generado. Cada posición de Bc cumple:

```
Bc(i,j) = Bnc(i,j) - ( KncRowRed[i] + BncRowRed[j] ) + KncRed;
```

Donde podemos apreciar que no hay dependencias de datos en cada posición i,j con posiciones adyacentes.

El kernel centerTestCUDA se encarga de realizar en paralelo estas operaciones, y podemos aprovechar que los datos están ya almacenados en la memoria de la tarjeta ahorrándonos la transferencia. El kernel es de la siguiente manera:

```
__global__ void centerTestCUDA(float* Bc, float* Bnc, float* KncRowRed, float* BncRowRed,
float KncRed, int M, int L){
int bx = blockIdx.x;
int by = blockIdx.y;
int tx = threadIdx.x;
int ty = threadIdx.y;
float valor = 0;

if (bx*BLOCK_SIZE+tx < M && by*BLOCK_SIZE+ty < L){
valor = Bnc[(by*BLOCK_SIZE+ty)*M + bx*BLOCK_SIZE+tx]
+ KncRed - (KncRowRed[bx*BLOCK_SIZE+tx] + BncRowRed[by*BLOCK_SIZE+ty]);
Bc[(by*BLOCK_SIZE+ty)*M + bx*BLOCK_SIZE+tx] = valor;
}
}
```

En la Figura 4.21 podemos apreciar como se construye la matriz Bc.

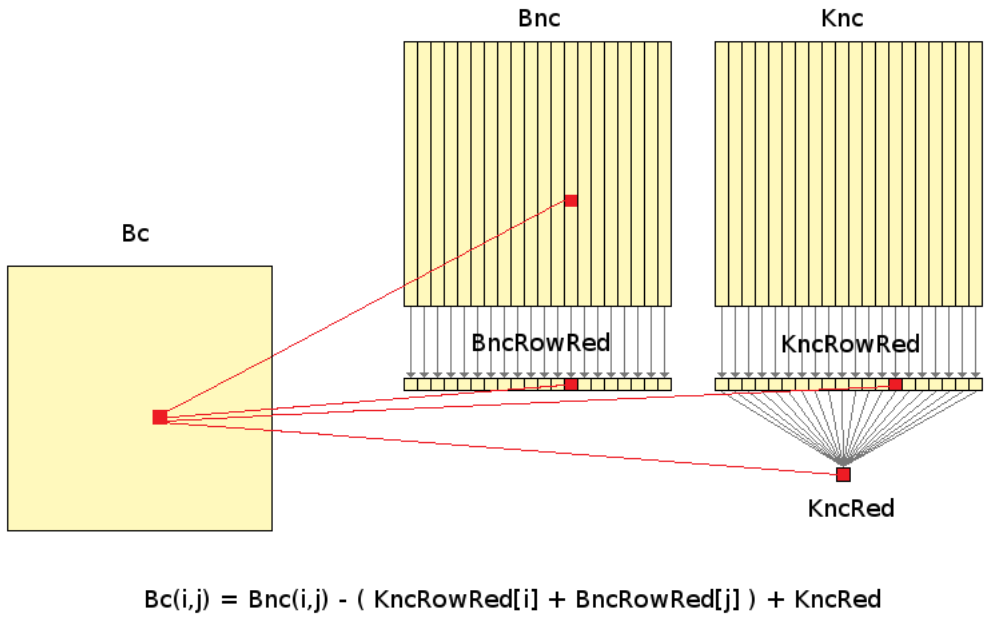


FIGURA 4.21: Cálculo de Bc en los kernel CenterTestGramMatrix

4.2.2. GramMatrix y KernelFunc

La función GramMatrix se encarga de generar la matriz Gram a partir de dos matrices m1 y m2 aplicando la función KernelFunc. Más concretamente cada posición (i,j) de la matriz Gram se calcula aplicando la función KernelFunc a la columna i de m1 y la columna j de m2. En cuanto a dependencia de datos se trata de una operación similar a la multiplicación, donde cada posición del resultado requería una fila de una matriz y una columna de la otra,

y vistos los resultados obtenidos de realizar la multiplicación en la tarjeta, es de esperar un comportamiento similar para esta función.

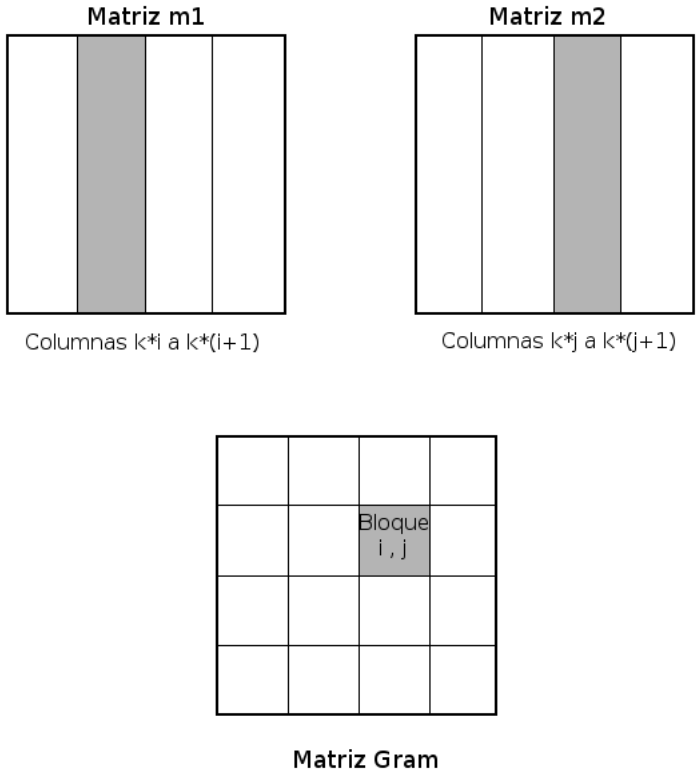


FIGURA 4.22: Datos necesarios para el bloque i,j de la matriz Gram

Crearemos una rejilla sobre la matriz resultado Gram, donde cada bloque de tamaño k*k necesita para calcular las posiciones que engloba k columnas de m1 y k columnas de m2 como se puede apreciar en la Figura 4.23.

Puesto que hay hilos que necesitan los mismos datos, tendremos que llevarlos a memoria compartida, para operar con ellos.

Una vez que se han traído los datos a memoria compartida, cada hilo aplicará la función KernelFunc a las columnas correspondientes de memoria compartida, y obtendrá el valor de esa posición de Gram. La Figura **??** muestra como se realiza la operación.

Dado que m1 es una matriz de gran tamaño, existe la opción de tenerla almacenada con una paleta P, donde creamos una escala de valores, y en la matriz m1 almacenamos índices a esa paleta. En caso de que m1 venga dada de esta forma, en el kernel hay una sección que se encarga de introducir en memoria compartida la paleta para realizar los cálculos adecuadamente.

4.2.3. Resultados Obtenidos

Gabor-KDA permite una identificación mucho más fiable como se puede apreciar en las curvas Roc de la Figura 4.24 y Figura 4.25. Por lo que es mucho más adecuado para realizar las proyeciones de las imágenes.

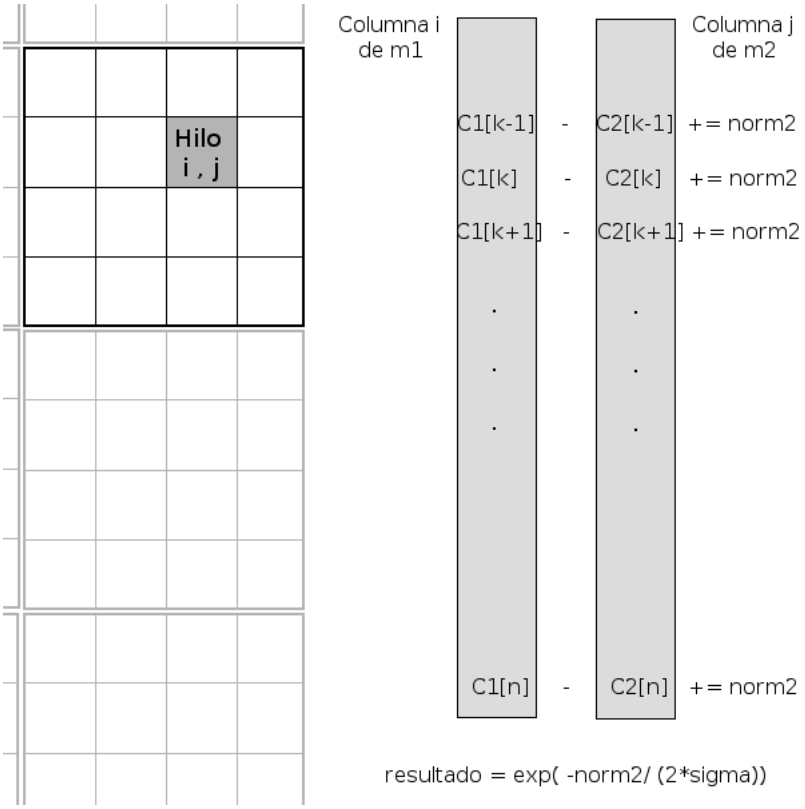


FIGURA 4.23: Cálculo realizado por cada hilo.

Hemos realizado la paralelización sobre el código con el que generamos la curva ROC, de la que obtenemos los valores con los que decidimos si las imagenes que proporcionamos son o no de la misma persona.

Las mejoras de rendimiento por lo tanto solo afectan a generación de dicha curva, y por ello no son tan significativas como los resultados obtenidos en NMF, donde las mejoras afectan a todo el proceso de entrenamiento, donde se realizan muchas más operaciones, y donde se aprovecha al máximo la capacidad de la GPU.

Concretamente para una base de datos de 93x99 imágenes de Cognitec [9], donde realizamos 9207 comparaciones para generar la curva ROC, el tiempo empleado sin usar la GPU es de 92 segundos, frente a 50 segundos que tarda usando la tarjeta, lo que supone un *Speed-Up* de 1.84.

Para realizar experimentos con bases de imágenes más grandes sería necesario disponer de una CPU con más memoria RAM,lo cual no ha sido posible para este proyecto. Aun así es de esperar que la GPU nos proporcione mejor rendimiento cuanto mayor sea la base de imágenes sobre la que trabajar.

[9]Empresa privada dedicada al reconocimiento facial.

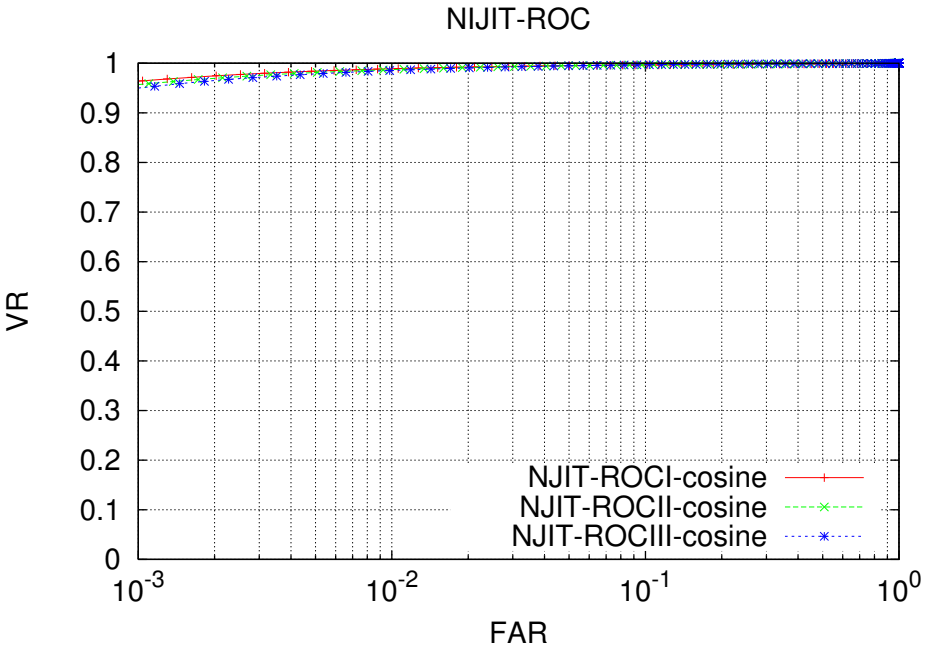


FIGURA 4.24: Curva Roc Experimento 1

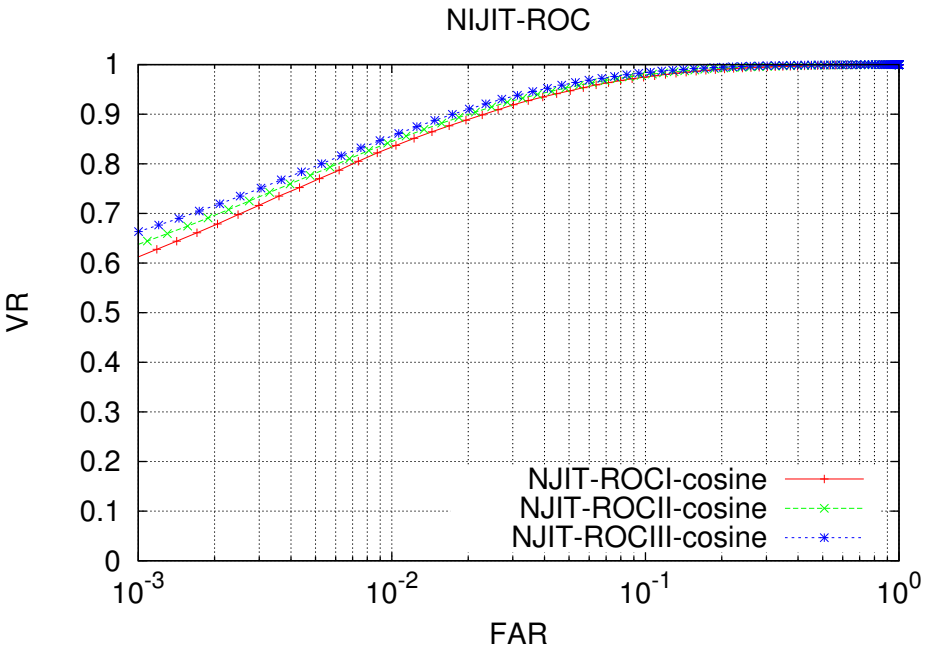


FIGURA 4.25: Curva Roc Experimento 4

Capítulo 5

Interfaz

Nuestra aplicación tiene un claro componente gráfico, al tratarse de reconocimiento facial a través de imágenes. La finalidad última es la de poder dar al usuario información clara y concisa sobre si dos fotos distintas corresponden a la misma persona o no.
El desarrollo ha sido bajo un sistema operativo Linux (Debian[1]), lo que nos obligó a tomar una primera decisión. En sistemas operativos de escritorio Linux hay dos entornos de escritorio que sobresalen del resto por su usabilidad y la riqueza de aplicaciones desarrolladas específicamente para ese entorno, que son GNOME[2] [GNO] y KDE[3]. Cada uno de estos entornos gráficos utiliza unas librerías gráficas distintas, siendo para GNOME GTK+ [GTK] y para KDE Qt.
Una de las principales diferencias entre estos entornos de escritorio son la filosofía de cada uno, siendo las aplicaciones para GNOME bastante más sobrias en cuanto aspecto, con controles sencillos, dejando muchas veces los controles avanzados para ficheros de configuración en formato texto. Al contrario que KDE, con aplicaciones que disponen de unos menús muy ricos y pueden configurarse más ampliamente desde la propia interfaz gráfica del programa.
Las razones por las que finalmente decidimos que nuestra aplicación tendría un entorno gráfico realizado con GTK+ se pueden resumir básicamente en que la sobriedad gráfica de GNOME le permite crear aplicaciones más ligeras, con menor consumo de memoria RAM, y también por una mayor familiaridad de todos los miembros del grupo con el entorno GNOME.

GTK+ son las siglas de The GIMP Toolkit, que es un conjunto de bibliotecas para desarrollar interfaces gráficas de usuario para GNOME entre otros.
Inicialmente estas bibliotecas fueron creadas para desarrollar el programa de edición de imagen GIMP[4] , pero posteriormente su uso se extendió a prácticamente todas las aplicaciones desarrolladas para GNOME y XFCE.
GTK+ está escrito en C, pero tiene bindings para lenguajes como C++, C# , Java o Python. Rápidamente nos decidimos por programar la interfaz en C, ya que los bindings de GTK+ para otros lenguajes no están completos y sobre todo el sistema de eventos es bastante mejor en C, además de tener una documentación más completa.

[1]Comunidad conformada por desarrolladores y usuarios, que mantiene un sistema operativo GNU basado en software libre precompilado y empaquetado, en un formato sencillo en múltiples arquitecturas de computador y en varios núcleos.
[2]GNOME son las siglas de GNU Network Object Model Environment.
[3]KDE son las siglas de K Desktop Environment.
[4]GNU Image Manipulation Program es un programa de edición de imágenes digitales en forma de mapa de bits, tanto dibujos como fotografías.

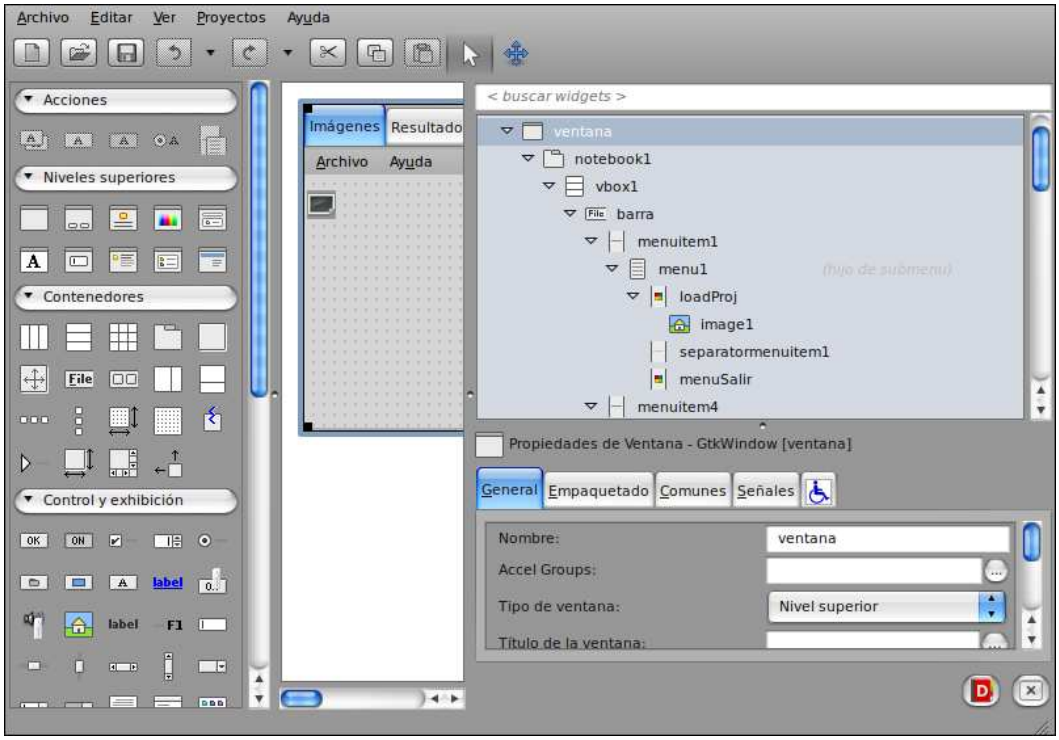


FIGURA 5.1: Vista de la ventana principal del diseñador de interfaces Glade

GTK+ contiene varias librerías para propósitos específicos, como ATK, que es la biblioteca para crear interfaces accesibles para personas con discapacidad, pero vamos a describir un poco más las bibliotecas que hemos utilizado para la creación de nuestro interfaz:

- GLib: es una biblioteca de bajo nivel de propósito general que se usa para implementar funciones no gráficas. Proporciona manejo de estructuras de datos para C, manejo de hilos, carga dinámica, etc.
- GTK: es la biblioteca que contiene los objetos y las funciones específicas para crear la interfaz de usuario. El tipo básico de objetos de esta biblioteca son los widgets, de los que heredan el resto de componentes.
- GDK: esta biblioteca es un intermediario entre gráficos de bajo nivel y gráficos de alto nivel, utilizado en nuestro caso para el renderizado de las imágenes en pantalla.

Glade [GLA] es un diseñador visual de interfaces gráficas para GTK+ que simplifica mucho el trabajo del diseño de las mismas. No genera código fuente sino un archivo XML que posteriormente es interpretado en tiempo de ejecución.
Glade es muy útil sobre todo para crear interfaces de forma sencilla y rápida, pudiendo además simplificar los sistemas de eventos de forma considerable.
En nuestro caso se hizo un boceto con glade de la interfaz gráfica que se fue modificando, y posteriormente el archivo XML cargado es mejorado mediante la programación en C. Glade no tiene un control de layouts para poder redimensionar widgets y mantener las proporciones.

Nuestra interfaz utiliza un modo de posicionamiento absoluto, es decir, la esquina superior izquierda de la ventana es la posición (0,0), y a partir de esa referencia, y obteniendo las dimensiones de los widgets, se pueden establecer proporciones entre ellos utilizando siempre medidas absolutas.

La interfaz gráfica de nuestra aplicación consta de una ventana con dos pestañas. La primera de ellas contiene una barra de herramientas donde se puede seleccionar el fichero proyector para usar en el algoritmo.
Debajo hay una estructura de dos paneles divididos verticalmente con componentes totalmente simétricos. Cada uno de estos paneles contiene un panel para poder visualizar una imagen que puede ser cargada mediante un seleccionador de ficheros. La imagen cargada puede recibir eventos de ratón, conociendo la posición exacta de la imagen donde se ha clickado. Esto se utiliza para poder pinchar sobre los ojos de la cara en la imagen y poder enviar las coordenadas a una función que procesa la imagen y la normaliza, quedando una nueva imagen que se mostrará debajo, a escala de grises, con 256 tonos de gris. Se mantienen en ejecución unas variables con la proporción de la imagen con respecto de la original para poder tomar como referencia las coordenadas del click de ratón sobre los ojos, ya que estas coordenadas se refieren a la imagen reescalada para caber dentro de la ventana. Esta normalización de las imágenes se lleva a cabo para poder compararlas, puesto que es con ellas con las que se lleva a cabo el algoritmo de proyección y comparación de imágenes.
Debajo de estos paneles con imágenes hay dos botones, uno para poder salir de la aplicación, y otro que realiza la comparación de las dos caras. Una vez cargado el proyector y normalizadas las imágenes, si pulsamos en este botón, la aplicación nos llevará a la otra pestaña, la de resultados.
En esta pestaña hay una división horizontal en dos paneles. El superior contiene una gráfica que nos sirve de guía para poder cambiar un parámetro que nos dará más o menos precisión a la hora de la comparación, y el inferior contiene una barra de deslizamiento para poder modificar este parámetro. Además, una vez realizada la comparación, aparecerá un texto indicando si son o no la misma persona.

Una vez ejecutada la aplicación, aparece la ventana principal. Lo primero que deberemos hacer es cargar un proyector, pinchando en "Archivo" dentro de la barra de herramientas, y posteriormente en "Cargar proyector", como podemos observar en la Figura 5.2. Aparecerá un selector de archivos, observable en la Figura 5.4 donde podremos elegir el proyector a utilizar. Hemos decidido que los proyectores entrenados con NJIT tengan extensión .proj, y los entrenados con NMF extensión .nmf. Esta diferenciación de extensiones entre distintos tipos de proyectores hace automático el proceso de elegir el tipo de proyección que necesitamos, ya que reconocemos automáticamente la extensión del proyector, y en el caso de NJIT no se modificarán las imágenes a comparar, mientras que en el caso de NMF debemos normalizar estas imágenes en un rango entre 0 y 1. Además el proceso de proyección es distinto para cada uno de los métodos.
Si no se ha cargado un proyector e intentamos comparar las imágenes, o simplemente queremos cargar una imagen, aparecerá un menú que nos indicará que primero debemos cargar el proyector, y que podemos observar en la Figura 5.3.
A continuación podemos proceder a cargar las imágenes, mediante otro menú de selección de archivos. Una vez cargadas las imágenes podemos proceder a clickear, primero en su ojo derecho, y posteriormente en su ojo izquierdo. Una vez hecho esto aparecerá la imagen normalizada más abajo en escala de grises. Si una vez cargado el proyector pero no cargadas las imágenes, o cargadas y no normalizadas, presionamos el botón comparar, el programa nos avisará mediante un menú de que primero debemos cargar y normalizar las imágenes.

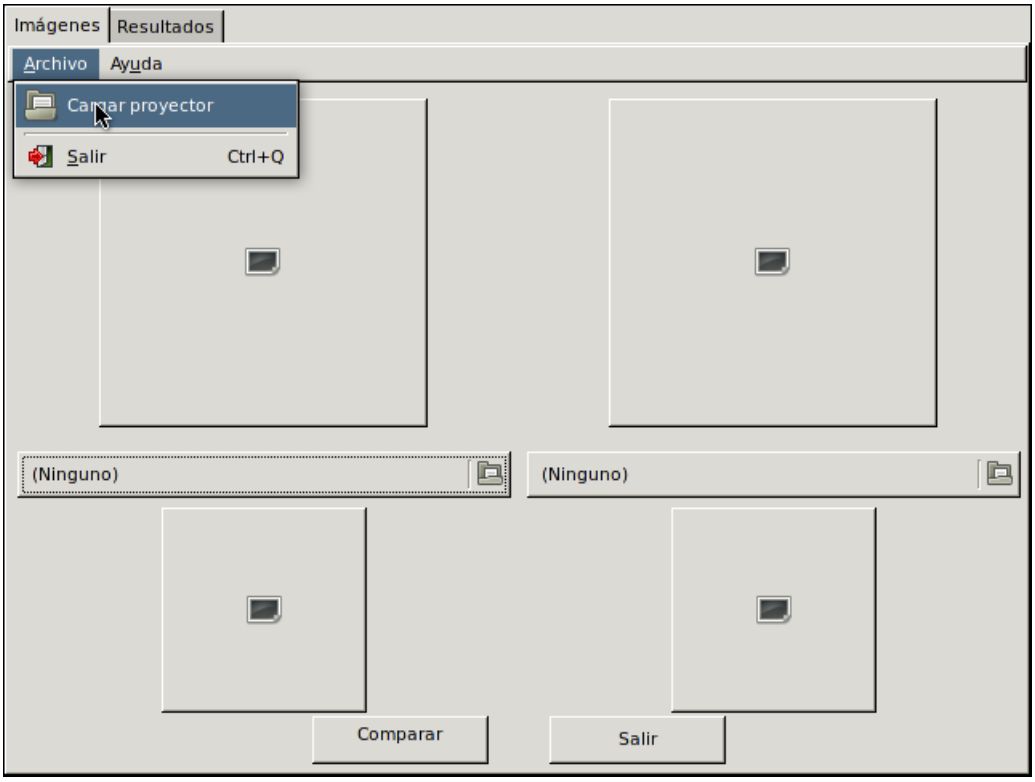


FIGURA 5.2: Menú de carga del proyector.

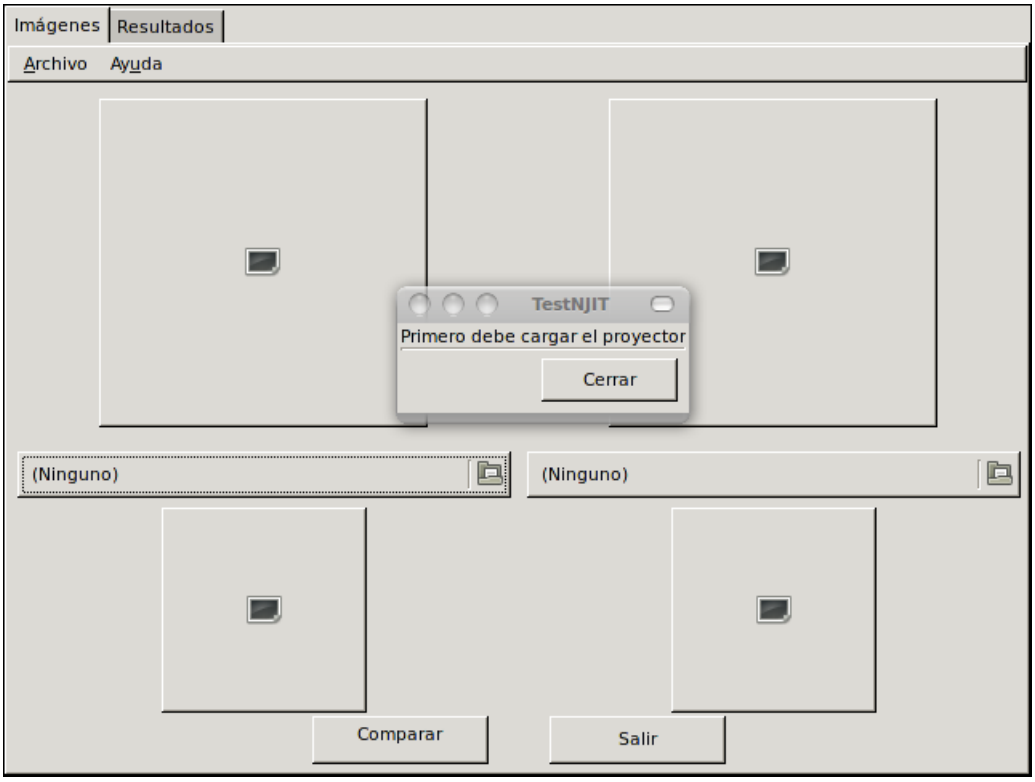


FIGURA 5.3: Advertencia por proyector no cargado.

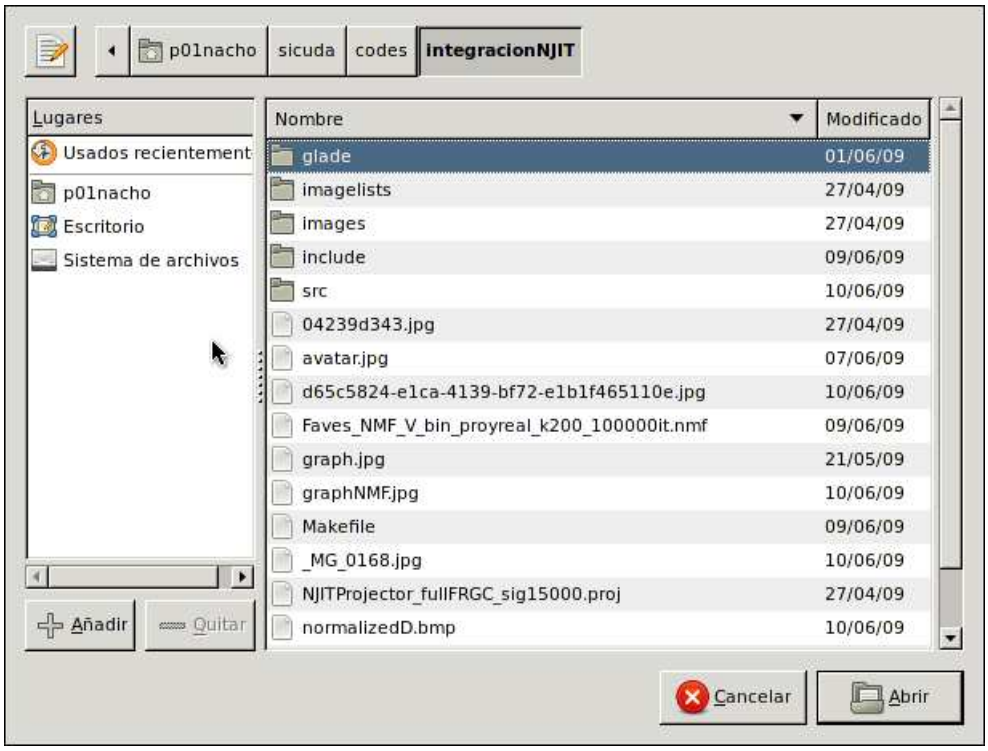


FIGURA 5.4: Menú de selección del proyector.

FIGURA 5.5: Fotos seleccionadas y normalizadas.

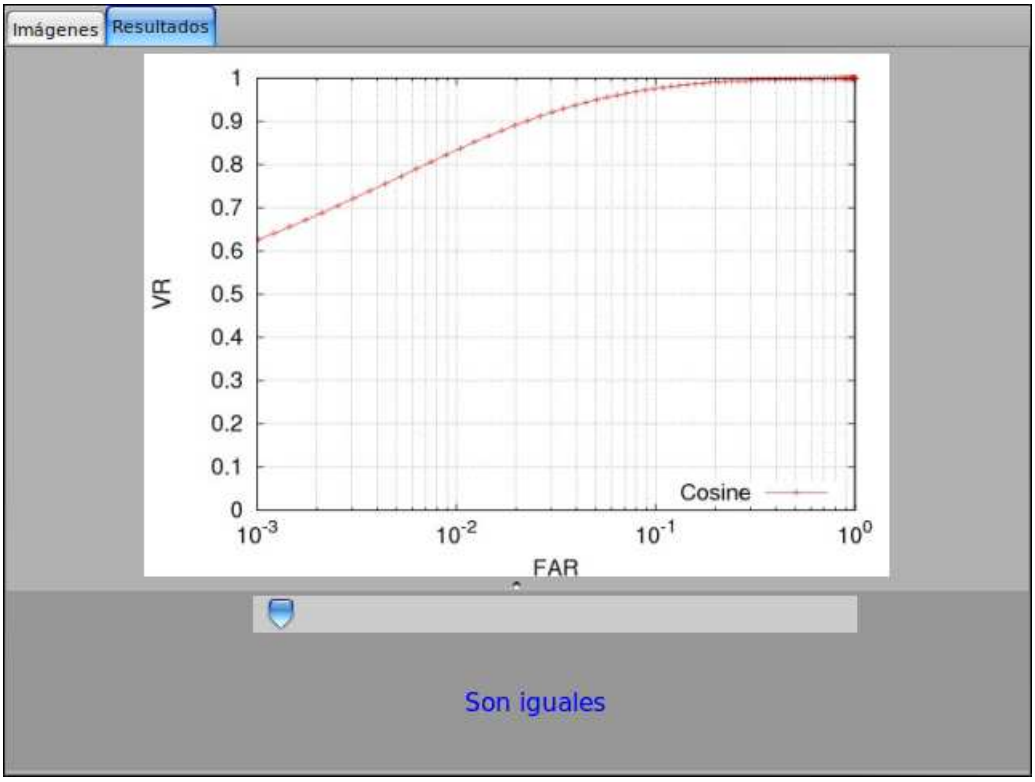


FIGURA 5.6: Menú de resultados.

Una vez cargadas y normalizadas las imágenes,como se puede observar en la Figura 5.5, podemos ir a la pestaña resultados, y arrastrar la barra para que se realice la comparación de imágenes de una forma más o menos estricta (cuanto más hacia la derecha menos estricta es la comparación, y más probabilidades de fallar). Una vez hecho esto, regresamos a la primera pestaña y presionamos el botón comparar, lo que nos llevará otra vez a la pestaña de resultados, donde aparecerá si las dos imágenes pertenecen a la misma persona o no, tal y como aparece en la Figura 5.6. En esta pestaña podemos volver a arrastrar la barra para ver cómo cambia el resultado de la comparación.

Capítulo 6

Conclusiones

Hemos desarrollado un programa con interfaz gráfica que permite la identificación de una persona contrastando dos fotografías. Para ello hemos seleccionado los algoritmos que han resultado más fiables tras la investigación previa, y que admiten paralelismo en sus operaciones más importantes, como son NMF y Gabor-KDA.

Los algoritmos para la identificación de las imágenes se han implementado sobre una GPU aprovechando el alto grado de paralelismo que ofrece. Uno de los aspectos más relevantes ha sido el particionado de los datos de modo que se minimicen las transferencias CPU-GPU puesto que son las más costosas, y dentro de la tarjeta hemos optimizado el uso de la memoria global y compartida que ofrece.

El factor tiempo es uno de los principales requisitos en el reconocimiento facial, y gracias al uso de la GPU hemos obtenido un rendimiento de hasta 200 veces superior a la versión paralela OpenMP en el algoritmo NMF.

Gabor-KDA sólo se ha optimizado en su parte *On-line*. Los resultados obtenidos no han sido tan buenos como con NMF debido a que los datos procesados son mucho más pequeños, y no permiten el total aprovechamiento de los recursos de la GPU. Aún así, sigue siendo muy interesante su uso, puesto que sus resultados en la identificación facial son mucho mejores que los de otros algoritmos, incluido NMF.

Posibles mejoras Entre los objetivos que han quedado fuera del alcance de este proyecto, sería muy interesante investigar las siguientes mejoras:

- El particionado de los datos y las transferencias CPU-GPU y GPU-CPU tienen una gran repercusión sobre el tiempo del programa. Nosotros hemos optado por un particionado estático, pero sería interesante que éste se realizase de forma dinámica con respecto al tamaño de las matrices que tengamos, incluso no realizando particionado alguno si la memoria de la tarjeta permite almacenar las matrices completas.
- La inicialización de los factores W y H del método NMF mediante *Nonnegative Double Singular Value Decomposition* (NNDSVD). Hemos realizado unas investigaciones preliminares con códigos en Matlab comparando la ejecución del algoritmo NMF con este tipo de inicialización, con otro con una inicialización aleatoria, y aunque la mejora en el tiempo no es muy significativa, la aproximación de V obtenida multiplicando W por H al final del algoritmo es mucho mejor.

Debido a la modularidad de nuestro programa, es posible ampliar sus funcionalidades para comparar imágenes con bases de datos, o incluir nuevos algoritmos que puedan ser de

interés.

Aunque los resultados obtenidos en NMF para reconocimiento facial no son suficientemente buenos como para realizar con él identificación facial fiable, por ejemplo en un control de accesos, hemos visto otros usos potenciales, por ejemplo en el campo de la bioinformática, realizando *clustering* de genes, o en la clasificación de sonidos, como en el caso de clasificar el sonido de una orquesta por instrumentos.

Bibliografía

[atl] Atlas project. http://math-atlas.sourceforge.net/.

[bla] Blas. http://www.netlib.org/blas/.

[Fis36] Ronald Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.

[FRG] Frgc. http://face.nist.gov/frgc/.

[Fri89] Jerome H. Friedman. Regularized discriminant analysis. *Journal of the American Statistical Association*, 84:165–175, 1989.

[GLA] Glade - a user interface designer for gtk+ and gnome. http://glade.gnome.org/.

[GNO] Gnome: The free software desktop project. http://www.gnome.org/.

[GPU] Gpu. http://www.nvidia.com/page/home.html.

[GTK] The gtk+ project. http://www.gtk.org/.

[Gui02] *Non-negative Matrix Factorization for Face Recognition*, 2002.

[Jol02] I.T. Jolliffe. *Principal Component Analysis*. Springer, Nueva York, Nueva York, EEUU, 2002.

[lap] Lapack. http://www.netlib.org/lapack/.

[Liu06] Chengjun Liu. Capitalize on dimensionality increasing techniques for improving face recognition grand challenge performance. *EEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 28(5):725–737, 2006.

[LS99] D. Lee and H. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401:788–791, 1999.

[Moo20] E. H. Moore. On the reciprocal of the general algebraic matrix. *Bulletin of the American Mathematical Society*, 26:394–395, 1920.

[NVIa] NVIDIA. Arquitectura geforce 280 gtx. http://www.nvidia.com/object/product_geforce_gtx_280_u

[NVIb] NVIDIA. Información sobre cuda. http://www.nvidia.es/object/cuda_what_is_es.htm

[NVIc] NVIDIA. Manual de programación en cuda version 2.1. http://developer.download.nvidia.com/compute/cuda/2_1/.

[Ope] Openmp. http://openmp.org/.

[Pen55] Roger Penrose. A generalized inverse for matrices. *Proceedings of the Cambridge Philosophical Society*, 51:406–413, 1955.

[Shl05] Jonathon Shlens. A tutorial on principal component analysis. *Electronic citation*, 2005.

[yKM99] S. Mika G. Rätsch J. Weston B.Schölkopf y K.R. Muller. Fisher discriminant analysis with kernels. *Neural networks for signal processing*, 9:41–48, 1999.

Índice de figuras

Índice de tablas